UNIVERSITY OF SOUTHAMPTON

Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

Evaluating Hardware Reliability in the Presence of Soft Errors

by

Bing Xue
MEng
ORCID iD: 0009-0000-4009-9277

A thesis for the degree of
Doctor of Philosophy

October 2024

University of Southampton

Abstract

Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

Doctor of Philosophy

Evaluating Hardware Reliability in the Presence of Soft Errors

by Bing Xue

Reliability has been a major concern in embedded systems. Higher transistor density and lower voltage supply increase the vulnerability of embedded systems to soft errors. A Single Event Upset (SEU), which is also called a soft error, can reverse a bit in a sequential element, resulting in a system failure. Simulation-based fault injection has been widely used to evaluate reliability, as suggested by ISO26262. However, it is practically impossible to test all faults for a complex design. Random fault injection is a compromise that reduces accuracy and fault coverage. Formal verification is an alternative approach. This research aims to utilize formal verification to evaluate the hardware reliability of a RISC-V Ibex Core in the presence of soft errors. We combine formal verification and fault injection, and perform backward tracing to identify and categorize faults according to fault effects (no effect, Silent Data Corruption, crash, and hang). With the help of formal verification, the entire state space and fault list can be exhaustively explored. We found that misaligned instructions can amplify fault effects. Apart from evaluating hardware reliability, the proposed method can help to determine a cost-effective fault protection strategy. We demonstrate how to use the method to formally evaluate the protection effectiveness of fault-tolerant technologies, for example, by identifying faults that can and cannot be detected/protected by fault-tolerant technologies. Formal verification, such as model checking, has limited applicability to Double Event Upsets (DEUs), due to infeasible runtime, proving efforts, and state explosion. We develop a DEU exploration strategy that significantly reduces model checking runtime and efforts to explore DEUs. We also mitigate state explosion using abstractions and proper constraints. As a consequence, we successfully scale our method to explore DEUs within an acceptable time. We found that DEUs can aggravate SEUs: a DEU consisting of two SEUs that cause no effect can cause a system failure.

Contents

List of Figures

List of Tables

Declaration of Authorship

I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published as: B. Xue and M. Zwolinski. Using formal methods to evaluate hardware reliability in the presence of soft errors. In 2022 17th Conference on Ph.D Research in Microelectronics and Electronics (PRIME), pages 29–32, 2022. doi: 10.1109/PRIME55000.2022.9816775

Signed:.. Date:..................

Acknowledgements

I would like to thank my supervisors, Professor Mark Zwolinski and Doctor Basel Halak, for providing guidance and feedback throughout this project. I am extremely grateful to my primary supervisor Prof. Mark, who was kind and patient to permit my suspension when I was ill. Thanks also to my parents for supporting me in finishing my study.

Definitions and Abbreviations

SEU	Single Event Upset
DEU	Double Event Upset
MEU	Multiple Event Upset
SV	SystemVerilog
SVA	SystemVerilog Assertion
TMR	Triple Modular Redundancy
DMR	Double Modular Redundancy
NMR	N-Modular Redundancy
SR	Shadow Registers
SEE	Single Event Effect
SET	Single Event Transient
SEFI	Single Event Functional Interrupt
SEL	Single Event Latch-up
EMI	Electromagnetic Interference
EMP	Electromagnetic Pulse
ISA	Instruction Set Architecture
SoC	System-on-Chip
IMEM	Instruction Memory
DMEM	Data Memory
SDC	Silent Data Corruption
RTL	Register Transfer Level
COI	Cone of Influence
PC	Program Counter
RHS	Right Hand Side
LHS	Left Hand Side
hart	hardware thread

Chapter 1

Introduction

Computer systems can be classified into two groups: general-purpose computers and special-purpose computers. A general-purpose computer, such as a desktop computer, can perform a range of different tasks. A special-purpose computer, which is also referred to as an embedded system, is designed for specific tasks. An embedded system is a part of another bigger device, such as a mobile phone, a vehicle, or a plane. Embedded systems are widely employed in electrical and mechanical devices. Unreliable embedded systems may lead to severe consequences [8]. For example, a malfunction that happens in a car can cause a car accident.

While simulation and emulation approaches reach their limits due to the complexity of the systems under verification, only formal proof techniques can ensure correctness according to the specification[9].

1.1 Reliability of Microprocessors

Technology scaling leads to better performance, lower power consumption, and higher transistor density. However, scaling of transistors and voltage significantly increases the vulnerability of embedded systems to soft errors[8, 10]. Soft errors are a type of fault. Researchers agree that soft errors have been a major concern of electronic reliability [11, 12, 13, 14]. Soft errors are also known as Single Event Upsets (SEUs). In this research, SEUs are the primary focus. Reliability is defined as the probability that the system under test will behave correctly, even in the presence of SEUs. Radiation events are the major causes of soft errors. A soft error occurs when an ionizing particle strike, which is caused by cosmic neutrons and alpha particles, reverses the state of a storage cell, such as memory, latch, and register [13]. In contrast to permanent errors, soft errors can be corrected if the storage cell is refreshed with new data. Radiation events in sequential logic result in soft errors, while radiation events in combination logic result in Single Event Transients (SETs). A soft error also occurs if an SET is latched into a storage cell [12, 13].

1.2 Cause and Effects of Faults

IEEE 1044-2009 considers faults a type of defect, and defines a defect as an imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced [15]. Two examples are given: 1) omissions and imperfections found during early life cycle phases and 2) faults contained in software sufficiently mature for test or operation. When expanded to the hardware area, a defect is the difference between the physical hardware and the original design. The term threat is used in safety analysis as a superordinate concept for three aspects: a fault is the initiator for misbehaviour, an error is the resulting misbehaviour in a system, and a failure is the visible effect of the fault [16].

A fault can be either a hardware defect or a software bug. This research focuses on hardware, hence only hardware faults are introduced. Ionizing radiation, electromagnetic effects, and electromigration are three major reasons causing hardware faults [17].

When ionizing particles strike semiconductors, free electrons and positively charged ions can be generated. The electrons can move and create a current, which may cause faults. For example, the created current alters data in sequential circuits (such as registers), leading to SEUs. These ionizing particles can be both internal and external. Internal particles are alpha particles due to alpha decay in the package materials and elements (such as uranium and thorium) in the fabrication process [18]. External particles such as protons, neutrons, pions, and muons are from background radiation like cosmic rays.

Electromagnetic effects such as electromagnetic interference (EMI) and electromagnetic pulse (EMP) can cause faults by inducing unwanted voltages and currents in the circuitry [19]. EMI is the disturbance of an electrical circuit due to the electromagnetic field of a nearby source; EMP is a short burst of electromagnetic radiation.

Various factors, such as increasing current flow and complexity, lead to hardware susceptibility to electromigration [20]. Electromigration is a process involving the net movement of metal atoms under the influence of electron flow and temperature, which can threaten hardware reliability [21]. While atoms are being driven from the cathode to the anode and vacancies are being driven in the opposite direction at the same time, if vacancy distribution is not at equilibrium, faults such as void and extrusion formation can occur [22].

Faults can lead to errors and failures. Avizienis gives a clear definition of faults, errors, and failures as well as the relationship, which are adopted in the following sentences[23]. A failure is an event when the performance of a system varies from the expected functions. When a failure occurs, at least one state of the system deviates from the correct state. The deviation is an error. A fault is the adjudged or hypothesised cause of an error. In other words, a fault may cause an error; multiple errors may cause a system failure.

Ziade also gives a clear definition of faults, errors, and failures [24]:

- A fault is a physical defect, imperfection, or flaw that occurs within some hardware or software.
- An error is a deviation from accuracy or correctness and is the manifestation of a fault.
- A failure is the non-performance of some action that is due or expected.

There are two types of faults: hardware faults and software faults. As the name implies, hardware faults occur at the physical level while software faults occur at the software level. Faults can also be categorized into two groups based on the duration of impacts: permanent faults and transient faults. Permanent faults are caused by physical component damage, such as short circuits or open circuits [24]. One solution to permanent faults is to replace or repair the damaged component. Transient faults will not cause permanent damage to hardware. One example of a transient fault is an erroneous state for a short time. After reviewing several pieces of research, Ziade concluded that transient faults occur more frequently than permanent faults and are harder to detect [24]. An SEU is a type of transient fault. The SEUs will be further discussed in Section 2.1.

1.3 RISC-V Ibex Core

RISC-V is a powerful open-source Instruction Set Architecture (ISA) designed for education, research, and industry [25]. There have been various cores and System-on-Chips (SoCs) that implement RISC-V ISA. For instance, Ibex Core. We have chosen the Ibex Core as the exemplar for this research, because the Ibex Core is an open-source 32-bit RISC-V CPU core that supports the basic RV32IMC instruction sets (Base Integer Instruction Set, Standard Extension for Integer Multiplication and Division, and Standard Extension for Compressed Instructions) [26]. In addition, it is

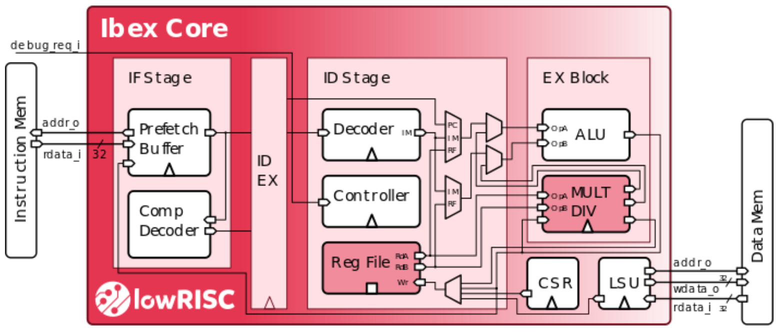


FIGURE 1.1: Ibex Pipeline from [26]

written in SystemVerilog, which supports a powerful formal language: SystemVerilog Assertion (SVA).

There are two pipeline stages in the Ibex Core, as shown in Figure 1.1. The first pipeline stage fetches and decompresses instructions from the instruction memory. The second pipeline stage decodes and executes the fetched instructions, and reads from and writes to the register file. One advantage of RISC-V is that many features are optional. The Ibex Core can be parametrised to enable multiple features. In this research, the Ibex Core is configured with the following features:

- Support RV32IMC (Base Integer Instruction Set, Standard Extension for Compressed Instructions and Standard Extension for Integer Multiplication and Division).
- A default simple core. No Physical Memory Protection, no Branch Prediction, no Secure Code Execution, no Instruction Cache, and no Lock Step.
- Disable all Interrupts and Debug Trigger Support.

There are 71 registers (2008 register bits) in this Ibex Core. This work focuses on the bits inside the core, i.e. all bits inside the rectangle titled 'Ibex Core' in Figure 1.1. The Register File is included, but the Instruction Memory (IMEM) and Data Memory (DMEM) are excluded. Memory safety, as such, is not an aim of this work. Moreover, there are various efficient memory error correction technologies and enhanced memory designs [27, 28, 29]. We assume that faults in IMEM and DMEM will be corrected within the memory and will not propagate to the core.

Ibex Core supports RISC-V Formal Interface(RVFI). RVFI interface captures useful signals which describe the behaviour of the core, such as executed instruction in the core, source register read data, and destination register write data. Such signals are useful for formal verification.

1.4 Cadence JasperGold

A formal tool is required in this research. We compared several open-source tools. We did not choose open source tools because: 1) most of them have limited support of SystemVerilog and SystemVerilog Assertions (such as PAT and PRISM), and 2) their limited functionality (i.e., PRISM cannot generate counterexamples). Therefore, we chose to use commercial tools. We compared some commercial tools: Questa Formal Verification Apps, Synopsys VC Formal, and Cadence JasperGold. All three tools are powerful. Considering our server configuration, we chose Cadence JasperGold as the formal tool. In principle, our method is general to all formal tools that support SVA. For open-source tools that do not support SVA, our developed properties would need to be transformed into another language.

1.4.1 Formal Property Verification

Cadence JasperGold is a platform containing a range of formal verification applications. In this research, the main used application is *Formal Property Verification* (FPV). FPV is a classic formal property verification application that is used to perform model checking. FPV takes user-developed SVA properties as inputs. Most SVA features are supported, such as sequence declaration and property declaration. A sequence is a list of Boolean expressions evaluated over time. A property is a set of design behaviours. Verification directives state what should be done with what behaviour. Supported verification directives include assert, cover, and assume. 'Assert' checks the property holds under all circumstances. 'Cover' demonstrates one example of how the property can be completed. 'Assume', which limits inputs to the DUT, defines constraints. Several useful SVA built-in functions are also supported, such as $past, $rose, $fell, $stable. In Appendix A we briefly explain some SVA syntax used in this thesis.

Figure 1.2 shows how FPV works. The formal technical details of model checking will be given in Section 2.4. The application takes constraints (i.e., SVA assumptions), specifications (i.e., SVA assertions), and the DUT (i.e., RTL in SV) as inputs. FPV first transforms everything into mathematical equations. Synthesis is involved in such a process, hence the design must be synthesizable. Then it analyses each specification (assertion) independently and tries to find an input sequence that causes wrong behaviour, violating the specification. If there exists such a sequence, a counterexample is found. If no counterexample can be found, the assertion is fully proved. If FPV cannot prove or disprove an assertion in a limited time due to complexity, the assertion is undetermined. Extra efforts, such as choosing different model checking techniques, are required to solve undetermined results.

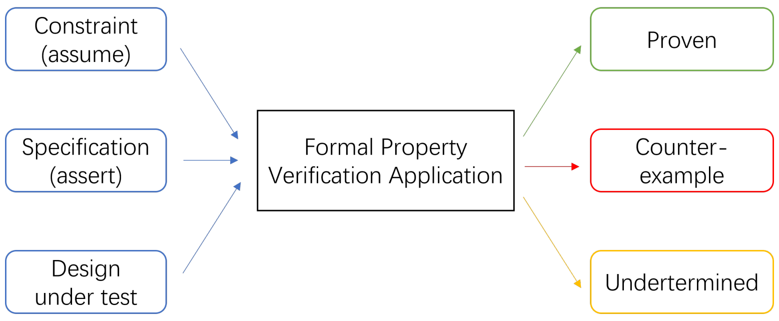


FIGURE 1.2: Process of Formal Property Verification

The verification process relies on Cone-of-Influence (COI). For each checker, FPV automatically determines all inputs, outputs and internal variables that influence the property. Such a collection is a COI of the checker. COI significantly reduces the verification complexity by removing unnecessary variables in the design. Perry gives a clear explanation of computing a COI [30]:

> Computing the variables contained in the cone of influence is straightforward. The tool begins by including the variables that directly occur in the property specification into the Cone-of-Influence set. Then for each variable contained in the cone-of-influence set, the tool will identify all variables that appear on the right-hand side of the RTL assignments to this variable and add the new variables to the COI variable set. This process is repeated until no new variables can be added to the set.

1.4.2 Functional Safety Verification

Other applications in the Cadence JasperGold platform are automated applications targeting particular verification tasks. Automated applications do not require user-developed formal properties. Formal properties are generated automatically by applications. For example, *Superlint App* checks design errors and coding errors that violate the coding style. *Functional Safety Verification* (FSV) can reduce the fault list in simulation-based fault injection by identifying and removing untestable faults. In addition, it can formally analyse fault propagation to improve fault classification. FSV duplicates a DUT (one good machine and one bad machine) and feeds the same inputs to both. Faults are injected into the bad machine. Strobes are inserted into the same locations (such as functional outputs and checker outputs) in the two machines. Then the strobes are compared to classify injected faults. The results can be propagated faults, unpropagatable faults, (always) detected faults and undetected faults.

There are three techniques implemented in FSV: Fault Relations Analysis, Fault Testability Analysis and Fault Propagatability Analysis. Fault Relations Analysis finds relations among faults, such as equivalence and dominance, to prune the fault list.

Fault Testability Analysis uses: 1) COI Analysis to identify untestable faults which are outside the COI but have a physical connection to the strobes; 2) Unactivatable analysis to identify unactivatable faults, such as a fault node permanently driven to a value; 3) Unpropagatable Analysis to find unpropagatable faults that cannot be observed on functional strobes.

Fault Propagatability Analysis uses: 1) Activation Analysis to identify safe faults which cannot be activated from inputs; 2) Propagation Analysis to check whether faults can (always) propagate to functional outputs; 3) Detection Analysis to check whether faults can (always) be detected by checker outputs; 4) Correlation Analysis to check whether propagated faults will always be detected.

FSV integrates both fault injection and equivalence checking to perform fault analysis. The results from FSV contribute to reducing simulation-based fault injection. The idea of using formal verification (such as equivalence checking) as a support for simulation-based fault injection to identify and classify faults has been utilized in many works, the details are in Section 2.5.

1.4.3 Proof Engines

Cadence JasperGold includes both SAT- and BDD-based engines with variations of these algorithms. For example, Engines B, C, C2, D, Hp, Ht, I, K, L, M, N, AD, AM use SAT solvers. Most of these well-known formal techniques/algorithms will be introduced in Section 2.4. We introduce the proof engines in this subsection.

In general, the proof engines can be divided into three groups based on proof objectives: 1) proving proofs exclusively; 2) searching traces; and 3) mixing of both. In addition, the proof engines can be divided into: 1) multi-property engines that analyse properties concurrently; and 2) single-property engines that analyse each property only once.

One of our experimental strategies in Chapter 3 is: finding all dangerous bits first, removing these dangerous bits from the fault list, and fully proving the remaining bits are safe. The former focuses on finding traces (counterexamples) and the latter focuses on full proofs. As a result, we started with the following engines, whose descriptions are in Appendix B. Some engines have both single- and multi-property versions, such as Hp and Hps. In that case, only single-property versions are introduced to save space.

Full proofs: engines N, Tri, Hp, Hps, AM, C, I, and R.

Counterexamples: engines B, Hts, Ht, AD, L, and U.

However, there is no 'best' engine mode which can solve all formal problems. Different engines are good at handling different problems. During experiments, we used different engine modes to solve different problems. We switched to a different engine mode if the current engine mode could not solve the investigated problems. We found some engines are useful and outperform the others. Details of these problems and suitable engines will be given in the experimental setups in Chapters 3 to 5.

1.5 Research Problem

The most common technology to mitigate SEUs is redundancy-based protection, either software redundancy such as SWIFT [31] or hardware redundancy such as Triple Modular Redundancy (TMR). However, redundancy of the whole design leads to excessive time and space overheads, which is a luxury for embedded systems. An alternative is partial protection, which only protects vulnerable structures with expensive protection technology while leaving reliable structures unprotected. The traditional method to evaluate reliability in the presence of soft errors is fault injection. Researchers [14, 32, 33] use fault injection to find the most vulnerable components in different processors and partially protect the most vulnerable components.

Simulation-based fault injection is widely used in reliability and safety research. ISO 26262 is an international standard of automotive safety that applies to all electrical, electronic, and software components [34]. ISO 26262 requires (random) fault injection to improve safety. To mitigate soft errors, researchers need to study the effects of soft errors and how soft errors are propagated in microprocessors. Researchers [12, 14, 35, 36] use fault injection to investigate soft error effects and propagation in microprocessors. There are two attributes of a fault: time and location. The injector in Figure 1.3 controls which clock cycle to inject a fault and which flip flop to inject the fault. The system under test is monitored for response to the fault. Researchers use fault injection to study the impacts of SEUs, assess the reliability of the systems under test, and evaluate soft error mitigation technologies. However, there are limitations to fault injection.

The first limitation is that it is practically impossible to test all faults. At most one SEU can occur in any bit at any time during one simulation. Considering the injection time and the injection location, there is an incredible number of possible combinations in the fault list. The simulation takes time. It is impossible to test all faults in the fault list

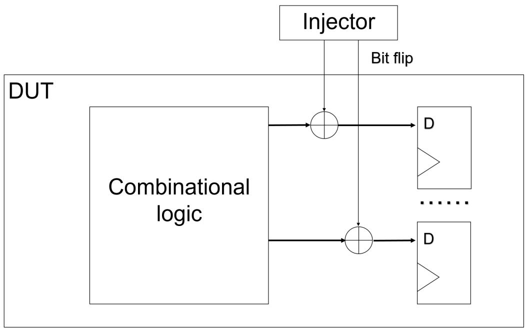


FIGURE 1.3: A classic fault injection diagram

within a reasonable time. As a compromise, only a small number of random faults in the fault list can be tested, which decreases the accuracy.

The second limitation is failing to cover all state space. Figure 1.4 shows a fault propagation path of fault injection. Each simulation can only cover limited state space. Even multiple simulations cannot guarantee to cover all state space, because corner states are hard to reach. As a result, passing all fault tests cannot guarantee the system is reliable.

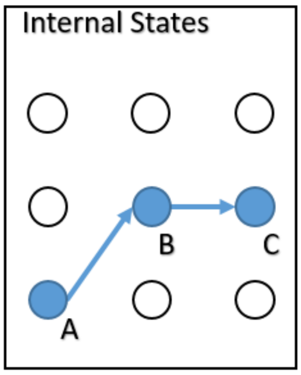


FIGURE 1.4: Fault tracing in a register net

The third limitation is that fault injection cannot perform root cause analysis of faults. Fault injection is cause-effect analysis: it injects a fault and monitors the response. It cannot identify/backtrack faults from an error or a failure. For example, with a wrong output caused by a fault, it is hard to use fault injection to find all candidate faults. We have not identified any recent successful research about root cause analysis of faults.

An alternative is using formal verification to perform fault analysis. However, the existing work focus on using formal verification to complement simulation-based fault injection, for instance, performing formal fault analysis to filter safe faults from the fault list, and then using simulation-based fault injection to test the remaining faults. The existing formal work only classifies faults into safe and none-safe faults.

There is no formal work that further explore non-safe faults based on the fault effects at the architectural level.

1.6 Research Hypotheses

Single Event Upsets (SEUs) in digital systems have been a major concern of reliability. The effects of such faults may be significant, but others will have no effect. We use errors/failures to denote Silent Data Corruption (SDC), crashes, and hangs. We call SEUs that may cause errors and system failures, such as SDCs, crashes, and hangs, crucial SEUs. We describe SEUs that result in no effects as safe SEUs. Similarly, we define crucial DEUs and safe DEUs.

We have the following hypotheses:

1. Our first research hypothesis is that we can identify and classify crucial faults. An analysis method will be developed to perform root cause analysis of faults. We propose developing formal properties to reveal faults that lead to SDCs, crashes and hangs. We propose using formal verification, such as model checking, to fully explore the entire state space, and to identify and classify SEUs based on the SEU effects.
2. Our second hypothesis is that we can mitigate the effects of SEUs by employing different methods. The proposed method will be expanded to evaluate fault-tolerant technologies. The evaluation results help determine a cost-efficient protection strategy to mitigate the effects of SEUs.
3. Our third hypothesis is that DEUs can aggravate SEUs. For example, DEUs that consist of two safe SEUs may lead to SDCs, crashes, and hangs. In other words, some safe SEUs are no longer safe in the context of DEUs. We aim to expand our method to DEUs. The generality of model checking to complex designs and DEUs is still limited because of infeasible runtime and state explosion. We can develop multiple complexity control approaches to make the runtime feasible and to make the results determined.

1.7 Research Objectives

Based on the hypotheses in Section 1.6, we have the following aims and objectives. Firstly we aim to formalize errors and system failures such as SDCs, crashes and hangs as formal properties. Formal verification will be used to explore all SEUs in the whole state space in the RISC-V Ibex Core. To ensure exhaustiveness, we aim to develop complexity control approaches and experimental strategies so that all the results are determined. By proving the developed properties, effect-cause analysis will be performed to find and categorize all safe SEUs that have no effect, and all crucial SEUs that lead to SDCs, crashes and hangs. The identified SEUs will be used to evaluate the reliability of structures (bits, registers and modules) in the microprocessor. We argue that some structures are more vulnerable than others.

Different hardware fault protection technologies have different costs and effectiveness. This research secondly aims to expand the proposed method to evaluate the effectiveness of fault-tolerant technologies. The evaluation results can be used to determine a cost-efficient protection strategy: use the most effective though the most expensive technology to protect the most vulnerable bits, use the less effective but cheaper technology to protect less vulnerable bits, and leave reliable bits unprotected. Redundancy-based fault-tolerant technologies have been verified by many works, so we aim to evaluate a different fault-tolerant technology named residue arithmetic.

This research thirdly aims to expand the proposed method to explore DEUs. We aim to identify and categorize all DEUs based on the fault effects. We aim to prove that DEUs can aggravate SEUs. We also aim to develop different approaches to scale model checking to DEUs, for instance, by reducing model checking time and efforts, and by mitigating state explosion.

The objectives of this research are outlined below:

1. Firstly, we aim to develop a formal method to perform SEU analysis.
    (a) To propose formal properties that reveal faults leading to SDCs, crashes and hangs.
    (b) To exhaustively identify and categorize all SEUs based on the fault effects by model checking the developed properties.
    (c) To develop various complexity control strategies to ensure exhaustiveness.
    (d) To evaluate the reliability of all hardware structures (such as bits, registers and modules) and software (such as instructions) in the presence of SEUs.
2. Secondly, we aim to extend the method to verify fault-tolerant technologies in the presence of SEUs.
    (a) To develop formal properties revealing the (raw and crucial) fault detection effectiveness of the implemented fault-tolerant technology.
    (b) To identify all hardware structures where injected faults can be detected by the implemented fault-tolerant technology.
    (c) To identify all crucial faults that can be detected by the implemented fault-tolerant technology (not all crucial faults can be detected).

(d) To evaluate the fault detection effectiveness and hardware cost of the implemented fault-tolerant technology.

3. Thirdly, we aim to extend the method to DEUs.

    (a) To mitigate state explosion caused by DEUs and to make the runtime and efforts feasible.

    (b) To identify and categorize all DEUs based on the fault effects.

    (c) To evaluate the reliability of all hardware structures and software instructions in the presence of DEUs.

    (d) To demonstrate that DEUs can aggravate SEUs.

4. We aim to demonstrate the above methods by applying them to a RISC-V Ibex Core.

1.8 Contributions

Contributions of this research include:

- We formalized three types of failures: SDCs, crashes and hangs as formal properties. By proving the developed properties, we identified and classified all SEUs that lead to SDCs, crashes and hangs. Compared to simulation-based fault injection, our method is exhaustive because all the state space can be covered. We applied various complexity control strategies to ensure exhaustiveness: all the SEUs in the fault list were tested and all the results were determined (except multiplications and divisions). Compared to other work that use formal verification to perform fault analysis and group faults into safe and crucial faults, our method can further categorize crucial SEUs into three groups based on the SEU effects. We chose the Ibex Core as our case study to evaluate the proposed method. Based on the experimental results, the method can evaluate the hardware reliability in the presence of SEUs. We found some hardware structures (bits, registers and modules) are more vulnerable to SEUs than others, similar to software instructions. We found FIFO and Program Counter are the most vulnerable structures in the Ibex Core. We found that compressed 16-bit instructions are more vulnerable than 32-bit instructions. We also found misaligned instructions can amplify fault effects. We validated the properties and the method with various approaches, such as simulation and mutations. We also used Triple Module Redundancy and Shadow Registers as framework validation measures. The properties can be adapted to other RISC-V processors with signal remapping. Hence, our method is, in principle, general to other RISC-V processors.

- We expanded our method to evaluate the fault detection effectiveness of fault-tolerant technologies. Different from other work that use formal verification to evaluate well-known redundancy-abased fault-tolerant technologies such as Triple Module Redundancy and Shadow Registers, we chose to use residue arithmetic as the exemplar. We developed properties to reveal all hardware structures where injected faults can be detected by the implemented fault-tolerant technology. We found not all the structures in the Ibex Core can be covered by residue arithmetic. It is meaningless to explore faults injected in the uncovered structures, hence reducing the fault space. We modified the properties in the last paragraph to identify all crucial faults that can be detected by the implemented fault-tolerant technology. We found not all the crucial SEUs in the last paragraph can be detected by residue arithmetic. We found that with little overhead, residue arithmetic can detect more than half of the crucial SEUs that lead to SDCs, and the detected crucial SEUs are in both the control and data paths in the Ibex Core. However, faults in the most vulnerable structures such as the FIFO and PC cannot be detected by residue arithmetic. The expanded method can identify all SEUs that can and cannot be detected by fault-tolerant technologies. The results help determine cost-efficient fault-tolerant technologies.
- We also expanded the method to evaluate the hardware reliability of the Ibex Core in the presence of DEUs. We exhaustively explored and categorized DEUs based on the fault effects. We identified vulnerable hardware structures and software instructions to DEUs, and some are more vulnerable than others. We proved that DEUs can aggravate SEUs, for example, two safe SEUs together can cause errors and failures. Hence, it is not enough to protect against SEUs only. There are few works about applying model checking to explore DEUs due to infeasible model checking time and state explosion. We developed various approaches to solve the problems, for instance, pruning the DEU list by removing DEUs that include crucial SEUs and identifying all safe DEUs in compact model checking runs. These approaches make it possible and time-acceptable to perform model checking to explore DEUs. State explosion is an inherent problem of formal verification. We developed multiple complexity control and experimental strategies to mitigate state explosion. These mitigation approaches greatly reduced model checking time and effort and ensured the results were determined.

1.9 Thesis Structure

This thesis is organized as follows. Chapter 2 introduces some technologies used in this thesis as well as literature reviews. Chapter 3 demonstrates the proposed formal method to evaluate the hardware reliability in the presence of soft errors. Chapter 4 extends the method to evaluate a fault-tolerant technology, residue arithmetic. Chapter 5 expands the method to DEUs. Chapter 6 is the conclusion.

1.10 Publications

A paper, as shown in Appendix C, has been accepted by PRIME 2022.

Chapter 2

Background and Literature Review

This chapter firstly introduces the background of this research, including Single Event Upsets, fault simulation and fault injection. Some simulation-based fault injection work is reviewed to explain the limitations of simulation. Then formal verification is discussed, including both methods and technologies. In addition, related work of using formal verification to perform fault analysis is reviewed. Finally, we compare the reviewed simulation and formal verification work to state the gap.

2.1 Single and Double Event Effects

Single Event Effects (SEEs) are device failures induced by single radiation events [13]. There are four types of SEEs: Single Event Upset (SEU), Single Event Functional Interrupt (SEFI), Single Event Transient (SET) and Single Event Latch-up (SEL) [13].

Single event upsets (SEUs), also called soft errors since they do not cause permanent hardware damage, have been the major concern in commercial terrestrial digital circuits [10, 13]. Ionizing radiation, electromagnetic effects and electromigration are three major sources of SEUs. Details of these causes have been explained in Section 1.2. For example, when a single energetic particle, such as an alpha particle from packaging materials or a neutron particle from cosmic rays, strikes a memory element and discharges itself. This causes enough energy to change the state of the memory element (a bit flip), and an SEU occurs. Apart from ionizing radiation, electromagnetic effects and electromigration, malicious attacks such as fault injection can also cause SEUs. SEUs occur in sequential circuits, hence directly influencing registers and memories in processors, for instance, a data change in registers. A characteristic of SEUs is that they are random events, and thus they may occur at unpredictable times [37]. The effects of an SEU (the altered data) exist until it is corrected or refreshed. Generally ionizing radiation, electromagnetic effects, electromigration and malicious attacks can only affect one bit at a time and cause an SEU. If multiple bits are influenced, Multiple Bit Upsets (MBUs) occur, which can lead to errors and failures even with error correction [38, 39].

An SEFI occurs if an energetic particle causes a bit flip in a critical system control register, resulting in system malfunctioning [40]. Different from an SEU, which may have significant impacts or may not, an SEFI leads to a system malfunction. An SET occurs if an energetic particle strikes the combinational logic, causing a voltage spike. If an SET is latched into a flip-flop, an SEU then occurs [41]. When an energetic particle passes through sensitive regions of a device, such as turning on a CMOS transistor between well and substrate, resulting in circuit malfunction, an SEL occurs [42]. SELs may cause permanent damage to hardware, such as short-circuits.

Some SEUs can cause errors and failures, while others do not. Some researchers [36, 37, 43, 44] performed fault injection to categorize SEUs into the following groups, based on the effects of SEUs

- Correct: The program in a processor completes normally. The output, the execution time and the internal states match with the golden run (fault-free run).
- Silent Data Corruption: The program in a processor completes normally without indication of errors. But the output differs from the golden run.
- Crash: The program in a processor crashes, and an error indication arises.
- Hang: The program in a processor fails to complete in limited clock cycles.

Only SEUs in the correct group have no effect. Others cause serious results. In this thesis, we use errors/failures to denote Silent Data Corruptions, crashes, and hangs.

2.2 Fault Simulation

Fault simulation is the most used method to test and simulate a faulty model of a circuit [45]. Fault simulation is very time-consuming because multiple faulty copies of the circuit as well as the original one are simulated multiple times [46]. Large designs, in particular when long test sequences are considered, also contribute to slow simulation [47]. Another reason is the limitations of simulation algorithms. There are three common algorithms used for fault simulation: serial, parallel, and concurrent [48]. All three algorithms require one or multiple copies of the design, which results in computational complexity. Fault simulation is a computationally intensive task. It needs to perform multiple simulations under different fault conditions, hence it is time-consuming. In addition, analogue circuits are increasing in SOCs; fault simulation in analogue circuits is even more time-consuming due to more complex algorithms and the combined need for more functional tests [49, 50]. There have been multiple methods to speed up fault simulation, such as behavioural fault modelling, symbolic simulation, parallelism, decision diagrams, and bitwise set operations [47, 49, 51, 52]. However, these methods cannot meet our requirements of exhaustively exploring all faults in the fault list, because it is still practically impossible to run exhaustive fault simulation.

Higher transistor densities and lower voltage supply increase the possibility of MBUs. Sangchoolie used a software-based fault injection technique to investigate whether multiple-bit flips can cause a higher ratio of Silent Data Corruptions (SDCs) than single-bit flips [44]. The fault model used is a bit-flip model, including single-bit flips and multiple-bit flips. Sangchoolie injected faults to both register files and pipeline registers at the instruction level [44], for example, reversing bits by modifying assembly code. 27,300,000 random fault injection experiments are performed, which can be categorized into three groups:

- Inject a single bit-flip
- Inject multiple-bit flips to the same register
- Inject multiple-bit flips to different registers

Experimental results show that multiple-bit flips do not cause much difference in the ratio of SDCs compared to single-bit flips. As a result, the single bit-flip model is enough for reliability studies. There is an incredible number of combinations of faults for single-bit flips. When it comes to multiple faults, the number of combination of faults is greater. He did not inject all combinations [44]. Instead, he used two methods to prune the fault space by only injecting faults to registers when the registers are used:

- Inject-one-read: Injecting a fault into a register before this register is read by an instruction. In other clock cycles, no fault is injected.
- Inject-after-write: Injecting a fault into a register this register it is written to by an instruction. In other clock cycles, no fault is injected.

These two methods significantly reduce the size of combinations by eliminating inactive registers. However, these methods rely on adding extra instructions to the program. The behaviour of the system, especially timing can be changed due to the extra instructions. Another drawback of Sangchoolie's work is injecting faults at the software level. Research has proved that software cannot access hidden registers in a microprocessor, thus losing accuracy [53].

To reduce the cost of fault simulation, Volk adopted program slicing to dynamic HDL slicing to prune the fault list [54]. Such a methodology can be divided into four steps. The first step is to build backward static slices for all microprocessor core outputs. A backward static slice is a set of all program elements that might affect (either directly or transitively) the values of the chosen observation point. It is similar to a Cone-Of-Influence (COI) in formal verification [55], which has been explained in Section 2.4. The second step is to run a simulation-based coverage analysis with a specific program/benchmark and record coverage data for each clock cycle. The third step is to compare the static slices with the coverage data to identify the intersection of both. The intersection is a dynamic slice. The final step is to select critical faults from dynamic slices and run simulation-based fault injection. Volk claimed that the methodology can successfully reduce the fault list by up to 10 percent. However, the methodology is program-dependent. With different programs, dynamic slices and hence critical fault lists are different. The worst case was only around 1 percent reduction. There are still too many faults for simulation. This methodology does not solve the problem of failing to cover all the state space and all faults.

2.3 Fault Injection

Faults could occur during all stages of an embedded system lifetime: specification, design, development, fabrication, installation, and operation. Single Event Upsets, which can reverse a state, have been a major concern for reliability, because they can cause system failures [11, 12, 13, 14]. Implementing SEU detection and recovery mechanisms during the design and development stages can significantly improve the system's reliability and reduce costs. Fault injection has been widely used to assess the reliability of a system; because it is cheap and accurate enough to the results [43]. Fault injection is a technique to assess the reliability of embedded systems. The basic idea of fault injection is to inject faults into the system and monitor the response of the system.

Different fault injection techniques can be classified into the following four groups:

- Hardware-based fault injection is performed in the actual hardware. Injection methods include power supply disturbance, heavy radiation strike, and beam testing [10, 56]. Compared to other types of fault injection techniques, hardware-based fault injection injects faults into the real hardware which executes the real software. As a result, this technique can cover all design faults that may occur in the actual operation time [24]. However, this technique may cause permanent damage to the hardware. It also has poor observability. Observers cannot check the testing results directly. Extra testing is required to evaluate the results. In addition, this technique has poor controllability. It is very difficult to control the injection location and the injection time.

- Software-based fault injection injects software faults at the software level to analyse the system [57]. This technique modifies the software, such as the assembly code, to emulate hardware faults. Compared to hardware-based fault injection, this technique does not require real hardware for the system, hence it has low cost and low complexity. In addition, this method can test errors in programs, which hardware-based techniques cannot. However, one drawback of this technique is that it cannot access and inject faults into hidden registers. Hidden registers are invisible from the software view, but they hold execution data temporarily. Laurent demonstrated the importance of hidden registers in the processor pipeline by injecting faults into a RISC-V Rocket processor [53]. What is more, this method modifies the program, usually by adding extra instructions. As a result, the original behaviour of the system may be influenced.

- Simulation-based fault injection injects faults into a simulation model of the system under test, and is usually VHDL or Verilog based [24]. The faults occur at the hardware level. This technique does not require a prototype, the system is simulated using software, such as Modelsim and Verilator. This technique can be performed at early design stages before the prototype is available to discover design faults and assess reliability. The biggest advantage of this technique is great controllability. Injection location, injection time, and fault duration can be controlled precisely. In addition, unlike the first two techniques, which can only observe the final output, this technique can monitor internal signals and states of the system. However, this technique is time-consuming. It can take a long time to simulate a complex system. The simulation time increases as the design size increases. The accuracy relies on the selected model. If the model is not accurate, the injection results may be useless. Considering the injection location and the injection time, there is an incredible number of possible combinations of faults. It is impossible to inject all the combinations. There are many states inside the system. Each simulation can only cover some of the states. Multiple simulations may cover some common states. Some states are hard to assess by simulation. Faults in these corner states can be regarded as untested if the corner states are not assessed during simulation. As a result, it is difficult to use this technique to simulate all combinations, and some injected faults may not occur in the real hardware. Though this technique is not as accurate as hardware-based fault injection, using accurate models can increase the accuracy [58]. With the same workload and system, the simulation-based fault injection technique is much faster and more accurate than the software-based fault injection technique [37].

- Emulation-based fault injection uses an emulator, such as an FPGA, to perform fault analysis. The target design, usually in VHDL and Verilog, is first loaded to an emulator. The injection process is then performed inside the emulator. The fault injection mechanism is usually separated into three parts: the emulated system, a fault injector, and a host. The fault injector, which injects faults into the emulated system, is controlled by the host. The host is either a computer connected to the emulator or an embedded CPU on the FPGA [24, 36]. This technique is much faster for injecting faults compared to simulation-based techniques. Multiple FPGAs can be used for emulation in parallel. For emulation, the initial system needs to be synthesized, placed & routed. One drawback of this technique is that the system source code must be synthesizable. The system source code for simulation-based techniques does not have to be synthesizable. Though this technique does not require a prototype, emulators are required. Though the emulation may be fast due to the high clock frequency of the FPGA, the speed of the whole fault injection process is limited by the communication speed between the host and the emulated system. For example, low data transfer speed between the host computer and the FPGA board can significantly increase the emulation time. Simulation-based fault injection can monitor the internal signals and states at any time. However, emulation-based fault injection can only check internal signals and states at a specific moment.

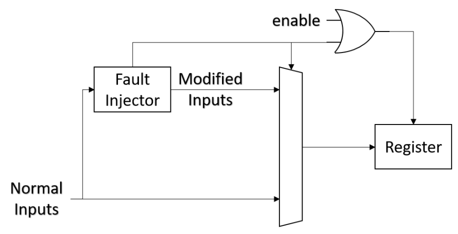


FIGURE 2.1: Fault Injection with a Mux

Touloupis injected both single-bit flips and double-bit flips to two LEON2 processors [17]. One processor has no fault tolerance (the default processor). The other triplicates the whole pipeline including pipeline registers and combinational logic. A voter is used to select the correct pipeline output. Three automotive benchmarks, mtx4x4, bitcnt, and qsort from MiBench are chosen as the workloads. MiBench is a free benchmark suite designed for embedded system tests [59]. Touloupis used a simulation-based fault injection technique. The LEON2 processors are implemented in VHDL (and cannot be synthesized). One advantage of using simulation-based fault injection is that all registers, including registers hidden from the software view, can have faults injected. A fault injector is designed to inject bit flips to destination registers at a specific clock cycle. Figure 2.1 is the fault injection mechanism implemented by Touloupis. The fault injector reads the fault list from a file before simulation. When reaching the target time, the fault injector flips the normal inputs, switches the multiplexer, and enables the target register. Such an injector mechanism can inject faults. However, it increases the complexity of the processors. Faults are injected during execution time. In other words, the program is not paused when injecting faults. So the execution time is not altered. Two extra modules are implemented: Execution Time Monitor and System State Monitor to generate a fault injection report. These two modules behave as their names imply and are not intrusive. Touloupis categorized multiple-bit flips into two groups: concurrent multiple-bit flips and non-concurrent multiple-bit flips. Results indicate that concurrent multiple-bit flips are not worse than single-bit flips. One special case is concurrent multiple-bit flips occurring in the same location, which is equivalent to a single bit-flip. The effects of non-concurrent multiple-bit flips are different. Touloupis used simulation-based fault injection to evaluate TRM of the whole pipeline and to compare the effects of multiple-bit flips and single-bit flips. Touloupis claimed that all internal registers, especially hidden registers are tested. However, Touloupis did not acknowledge that some states, hence registers, are hard to reach using simulation. In addition, Touloupis used random fault injection. Random injection cannot guarantee all faults are covered.

Cho injected single-bit flips into registers in two RISC-V processor cores to investigate soft error effects [36]. These two processors have different microarchitectures: one processor is in-order (Rocket), and the other is out-of-order (BOOM). FPGAs are used to emulate the processors and to inject faults. The target processors are designed using Chisel. However, the Rocket Chip Generator can convert the source code to Verilog. The generated Verilog net-list is modified to add a fault injector. Figure 2.2 is the fault injection mechanism used by Cho. As shown in Figure 2.2, each flip-flop in the emulated processor is driven by an XOR gate. The inputs of each XOR gate are the original input (the correct bit from the combination logic) and the control signal from the fault injector. As shown in Table 2.1, the input bit to the target flip-flop is flipped only when the control signal is one. The fault injector is controlled by the host CPU embedded in the FPGA. The host CPU selects the target flip-flop and the injection time randomly. The results of fault injection to the target processors reveal a strong correlation: soft error rates of the same application in different microarchitectures but the same ISA processors have a linear relationship. Based on this observation, Cho developed a method to predict the soft error rate of a target processor using soft error effects established on a known processor that has the same ISA as the target processor. The prediction error is within 7%. One advantage of this method is using XOR gates to flip bits. Some researchers implement multiplexers, such as [17], where the fault injector reads a bit in the target flip-flop, reverses the bit, outputs the reversed bit, and enables the target flip-flop and multiplexer at the same time. Compared to using multiplexers, using XOR gates simplifies the fault injection structure. There are many flip-flops in a processor. It is costly to pick up these flip-flops manually. Another advantage of this method is using Synopsys Design Compiler to generate a list of flip-flops in the target processor automatically. One drawback of this method is random fault injection. Cho injected about 1.6 million faults, but some target flip-flops may not be used during the emulation. In other words, the injected bit is not used. It is unclear how many fault injection experiments are meaningless. The BOOM core is more complex than the Rocket core. There are more flip-flops in the BOOM core than the Rocket core. However, Cho injected the same number of faults into both processors, which may influence the accuracy of soft error rates. Cho found strong correlations between the fault injection results from the two cores. Based on this finding, Cho argued that it is possible to predict the reliability of processors with the same ISA. However, the conclusion is based on RISC-V. Other ISAs are not tested.

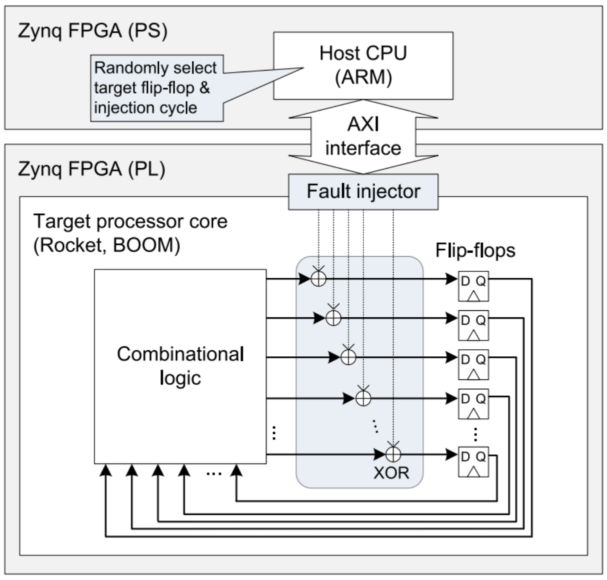


FIGURE 2.2: Fault injection mechanism used in [23]

TABLE 2.1: Truth table of the XOR gate

Correct Input	Control Signal	XOR Output
0	0	0
0	1	1
1	0	1
1	1	0

Ramos used fault injection to characterize the SEU sensitivity of lowRISC, a RISC-V processor [43]. The target processor is implemented in an FPGA. So, this is emulation-based fault injection. A single bit-flip is used as the fault model. Unlike other research, which inject faults into a normal memory that stores execution data and corrupts the stored data, Ramos injected single-bit flips into the configuration memory in the FPGA. A bit flip occurring in the configuration memory changes the design or the routing, because the configuration memory controls how the logic maps in the FPGA. For example, a bit flip in the configuration memory may change an OR gate into an AND gate. A Soft Error Mitigation (SEM) Core is used to inject faults into the emulated processor. SEM is a tool developed by Xilinx to perform SEU detection, correction, and classification for configuration memory. SEM also injects faults into the configuration memory [60]. Results show that less than 10% of the faults affect the execution. The injection location is selected randomly. There are more than 9 million configuration memory bits in the FPGA board. The randomly selected bits are a small part. As a result, there is an argument about the accuracy of the final results. One interesting point of Ramos's work is using an existing tool (SEM) to inject faults. Using SEM saves efforts to develop a new tool. However, SEM does not inject faults in real-time. SEM pauses the execution when flipping a bit in the configuration memory. Using SEM delays the execution time. There is a type of failure called ”failing to finish in time”. SEM cannot identify this type of failure.

Travessini used fault injection to analyse SEU effects in processor cores [14]. Travessini injected single-bit flips into registers in a LEON3 processor. Simulation-based fault injection is used in Travessini's work. Built-in simulator commands are used to automatically inject faults and monitor effects. The core module in the LEON3 processor is duplicated. The copied core module receives the same inputs as the default one. Outputs of the copied core module are disconnected from the rest of the system. As a consequence, the copied core does not alter the processor operation. One core is used for fault injection, the other is the reference. The interfaces of both cores are monitored in order to investigate the fault propagation. Results show that only a third of the total injected faults propagate to the core interfaces. The whole time for one simulation can be divided into two phases: the warm-up phase and the execution phase. The major work in the warm-up phase is to load the program into the program memory. The execution phase is the actual simulation phase. In order to analyse SEU effects in hardware, researchers should inject faults during the execution phase. However, the time of the warm-up phase varies for different programs and different designs. This work just considers the final 80% of the whole time as the execution time, which is not accurate enough. Some faults may be injected during the warm-up phase, hence reducing the accuracy of results. Another disadvantage is random fault injection. The third disadvantage is duplicating the whole core. Duplicating the core increases the complexity of the system under test and doubles the number of fault injection locations. This may be unacceptable for some large and complex processors.

Traditional fault tolerance techniques are based on redundancy. Dual Modular Redundancy (DMR) and Triple Modular Redundancy (TMR) are well-known hardware redundancy-based techniques. The basic idea of N Modular Redundancy (NMR) technique is replicating the default design N times. Inputs to all copies are the same as the inputs to the default system. A voter is used to select the output. NMR techniques result in more than N times overhead. To reduce the overhead, researchers choose to protect the most vulnerable part of the system. Such a technique is called partial protection. The most difficult question is how to find the most vulnerable part.

Many researchers use fault injection to select the most vulnerable part. In order to find the most vulnerable pipeline registers, Jeyapaul proposed a methodology to quantitatively analyse the vulnerability to soft errors of pipeline registers [32]. By injecting SEUs into pipeline registers in an AMBER core, which is an open ARM-v2A processor core designed in Verilog, Jeyapaul discovered that pipeline register vulnerability depends on instructions. In other words, some bits involved (and hence are vulnerable) in one instruction may not be used (are hence reliable) by another instruction. Based on this finding, an ISA-aware systematic methodology is proposed to analyse pipeline register vulnerability. There are two steps in the methodology. The first step selects the active pipeline register when executing an instruction. The second step computes pipeline register vulnerability. The results show that some of the control bits and some of the instruction bits contribute most to the system's vulnerability. By protecting some specific instruction and control bits, the vulnerability of the system can be reduced by 31% with 3% power overhead. The most interesting contribution of this work is ISA awareness. By injecting faults into pipeline registers and comparing the results of executing different instructions, Jeyapaul discovered that the pipeline register vulnerability depends on the instruction executed. Other researchers have not considered this. Jeyapaul called the vulnerability with ISA awareness the effective vulnerability. The effective vulnerability is only 12% of vulnerability without ISA awareness. This work uses a simulation-based fault injection technique. The injection time is selected randomly, but the injection target is not. It is unclear whether Jeyapaul considered the warm-up phase or not. RTL behavioural analysis is performed first to identify the active pipeline registers. Some bits identified by RTL behavioural analysis are not actually vulnerable. Fault injection is then used to verify the vulnerability of the identified bits using simulation. Performing RTL analysis to filter inactive registers is a smart idea. However, Jeyapaul did not give details about this process. It is also unclear if Jeyapaul considered when to inject faults into the identified bits. If the faults are injected in the warm-up phase, or after the bit is no longer used, the target bit may be recognized as the reliable bit, hence reducing accuracy. Jeyapaul implemented an access tracker in Gem5 to calculate vulnerability. One question is that AMBER uses ARM-v2A, which Gem5 does not support. Jeyapaul did not explain how to solve the mismatch. This may reduce the accuracy of the vulnerability analysis. Vulnerability analysis and partial protection in this work is instruction-specific. Embedded processors always execute specific programs. As a result, the ISA awareness analysis and partial protection can be a good methodology to improve reliability.

Plusquellic proposed an emulation-based hardware-software co-design fault injection method [61]. Different from others, Plusquellic's method addresses hardware reliability from a security perspective. This method only identifies faults that result in sensitive information leakage or corruption. RISC-V Rocket is chosen as the study case. The method with the Rocket core is emulated on an FPGA board. Plusquellic performed exhaustive fault injection and observed some security holes. Based on the results, Owen proposes a countermeasure [62]. Plusquellic's work focuses on security and hence focuses less on reliability. Plusquellic's methodology requires a huge number of fault injection experiments for different benchmarks (over 5 million experiments are performed for two benchmarks).

In summary, fault injection is widely used to assess the reliability of an embedded system. Fault injection injects faults into the system under test to monitor the response. There are two attributes of faults: when to inject faults and where to inject faults. Most researchers choose injection targets and injection time randomly, leading to fewer possibilities but less accuracy. Some researchers slightly reduce the size of fault combinations by fault pruning. Compared to the total combination of faults, the injected faults are only a small group. Passing fault injection analysis cannot guarantee the reliability of the system. In addition, fault injection-based analysis is forward tracing. It can trace the fault propagation in the system, but it cannot find the root cause of faults in the presence of a system failure.

Due to the infeasible time consumption, it is practically impossible to simulate all faults. Some techniques are developed to prune the fault list, such as static analysis and dynamic slicing [54]. Some researchers use hardware emulation to speed up fault injection, such as using multiple FPGAs [36]. However, it is still impossible to test all faults. Most researchers choose to inject a small number of faults, which reduces efforts with the cost of decreasing accuracy.

2.4 Formal Verification and Model Checking

Formal verification uses mathematical methodologies to verify whether the design under analysis meets design specification or not [63]. Kropf defines formal verification as the following [64]:

> A formal verification is a concise description of the high-level behaviour and properties of a system written in a mathematically-based language.

Formal verification can be applied to different stages in the design flow. Figure 2.3 shows two formal verification approaches applied to different design states. There is a concern that after synthesis and place & route, designs are not equivalent. For example, an AND gate is changed to an XOR gate by accident. It is difficult to debug such differences manually. Equivalence checking compares the outputs of two different models, such as RTL & gate level, with the same sets of inputs, to prove designs are equivalent [65]. Equivalence checking explores and compares all reachable state spaces. Compared to exhaustive simulation, equivalence checking (if tractable at all) will often be faster. One drawback of equivalence checking is that it heavily relies on the exploration and comparison of the reachable state space [66]. If some states are not reachable or not compared, the results of equivalence checking are not accurate.

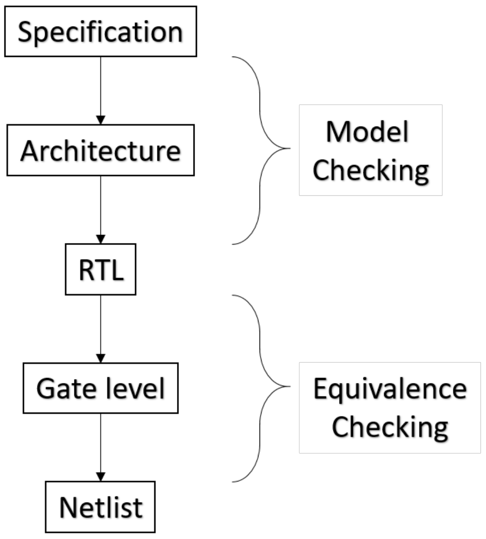


FIGURE 2.3: Formal Verification in ASIC design flow

This research mainly uses model checking. In the following subsections, we will introduce some well-known model checking techniques involved in Chapters 3 to 5.

2.4.1 Overview of Model Checking

Model checking is a formal method to analyse dynamic systems which can be modelled by state-transition systems [55, 67]. Figure 2.4 shows an overview of the model checking flow. Both design descriptions and design requirements are translated into mathematical models before they are verified by a model checker.

2.4.1.1 Modelling

The design is modelled to a state-transition system. State-transition systems are basically labelled graphs where nodes represent states, and edges model transitions [55, 67]. For instance, a Kripke structure [68]. Let AP be a set of atomic propositions. Atomic propositions are the fundamental parts of propositional logic and cannot be divided into smaller parts. Atomic propositions can only be true or false. A Kripke structure over AP is a triple $K = \langle S, T, L \rangle$, where S is the set of states, $T \subseteq S \times S$ is the transition relation, and $L : S \to 2^{AP}$ is a labelling function that gives the set of atomic propositions true in each state.

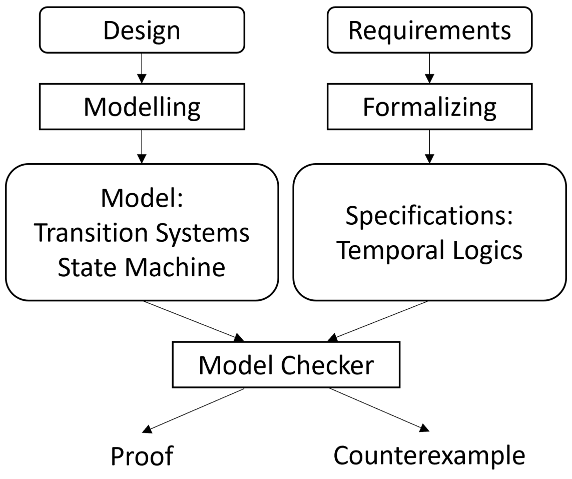

Design
Requirements
Modelling
Formalizing
Model: Transition Systems State Machine
Specifications: Temporal Logics
Model Checker
Proof
Counterexample

FIGURE 2.4: Overview of model checking flow

Another example of a state-transition system is a symbolic state machine in a synchronous hardware circuit: a state represents the current value of the registers together with the values of the input bits, and a transition models the change of the registers and output bits on a new set of inputs. Most designs under test are not in the form of explicit state-transition systems. Model checking tools can transform them into state-transition systems, similar to formal properties.

2.4.1.2 Formalizing

The design requirements are formalized into property specifications. Temporal logics, such as Linear Temporal Logic (LTL), Computation Tree Logic (CTL) and a combination of both (CTL*), are normally used languages to formally express formal properties [55, 67]. The time view in temporal logic can be either linear or branching. In LTL, every point in time has only one successor. Extra LTLs are required to handle branches, because computation paths from a branch are considered separately. LTL describes events along a single computation path. In CTL, branches are captured, so there may exist multiple successors for a given point. A tree structure can be generated to denote all computation paths.

The formal properties can be either safety properties or liveness properties. Safety properties state something bad will never happen, liveness properties state something good will eventually happen. Liveness properties should be used with caution because it may take a lot of time and effort to prove them but get meaningless results. For convenience, only safety properties are involved and discussed in this chapter. For example, a safety property (also called an invariant) can be expressed in CTL as AGp: the property p globally holds in all reachable states along all paths.

2.4.1.3 Model Checker

A model checker checks whether the model satisfies certain property specifications. If not, a counterexample will be given, such as a witness (path) for the satisfaction of $\neg AGp$. Another example is a waveform including all related inputs and state transitions violating the formal properties.

The basic algorithms of CTL model checking is: given a set of atomic propositions AP, a Kripke structure $K = \langle S, R, L \rangle$ and a CTL formula f, find all states s in K that satisfy f and check if initial states are included to satisfy f.

There are many different approaches/algorithms for the implementation of a model-checking procedure. One historical approach is Explicit-State Model Checking. As the name implies, the state descriptor for a system is maintained in explicit instead of symbolic form, along with all state transitions. For example, using a depth-first search algorithm to explore all state space and state transitions explicitly. Though some abstraction techniques and partial-order reduction algorithms are used to reduce the state space, Explicit-State Model Checking still suffers from state exploration.

2.4.1.4 Abstraction and Reduction

Abstraction and reduction are two techniques to mitigate state explosions. Abstraction reduces a system model to a smaller over-approximation model. if a property holds for the abstract model, then the property also holds for the original model. One common abstraction technique for hardware verification is localization abstraction [55, 69]. For a given property, localization abstraction only considers the logic necessary to prove the property; hence the complexity can be reduced by removing some unnecessary logic. For instance, abstracting a counter value. Partial-order reduction is effective for asynchronous systems [55, 70]. Considering an asynchronous system with n concurrent transitions. Then there will exist $n!$ different orderings and $2n$ different states (each transition has two states). However, many properties do not care about the ordering, hence it is sufficient to consider only a few sequences.

2.4.2 BDD and BDD-based Model Checking

An alternative model checking technique is Binary Decision Diagram (BDD)-based symbolic model checking [71]. A BDD is a data structure for representing and manipulating Boolean functions in symbolic form, such as graphs representing Boolean functions [72]. BDD is canonical and compact compared to explicit representations of states and transitions, because BDD can consider a large number of states as a state set and operate on the state set instead of individual states. A simple circuit with three XOR gates in Figure 2.5 is used to demonstrate how to construct a BDD in hardware verification.

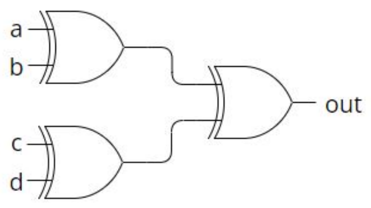


FIGURE 2.5: A simple circuit with three XOR gates

The first step is to build a truth table, as shown in Table 2.2. Then a Binary Decision Tree (BDT) can be generated, as shown in Figure 2.6. The top node labelled with a is a root. The bottom nodes (0s and 1s) are leaves. The circuit in Figure 2.5 can be formulated as a Boolean function with four variables (inputs), hence the depth of the BDT is four, and there are 2^4 possible leaves and paths. In the BDT, each node is labelled with a variable and has two outgoing edges. A solid line represents a 1-edge and a dashed line represents a 0-edge. Different ordering of variables may change the BDT. Ordering is important in a BDD. We choose an arbitrary order $a < b < c < d$ for simplification. An Ordered Binary Decision Diagram (OBDD) is an ordered BDT. A Reduced Ordered Binary Decision Diagram (ROBDD, simply called BDD) is the compact version (removing all redundancies) of OBDD.

TABLE 2.2: Truth Table of Figure 2.5

a	b	c	d	out	a	b	c	d	out
0	0	0	0	0	1	0	0	0	1
0	0	0	1	1	1	0	0	1	0
0	0	1	0	1	1	0	1	0	0
0	0	1	1	0	1	0	1	1	1
0	1	0	0	1	1	1	0	0	0
0	1	0	1	0	1	1	0	1	1
0	1	1	0	0	1	1	1	0	1
0	1	1	1	1	1	1	1	1	0

Then BDDs can be generated from BDTs. The BDT in Figure 2.6 is generated from the truth table and is not compact at all because it contains redundancies. The third step is to reduce redundancies. Hu summarized the reduction process[73]: 1) obeying the restrictions that along any path from root to leaf, no variable appears more than once, and that along every path from root to leaf, the variables always appear in the same order; 2) merge duplicate/isomorphic nodes; 3) delete redundant tests. The yield directed acyclic graph in Figure 2.7 is the BDD for the circuit in Figure 2.5.

BDD image computation is a basic in model checking. Suppose in a system, there is a set of variables V, the current reachable states are S_c, the next states are S_n and a

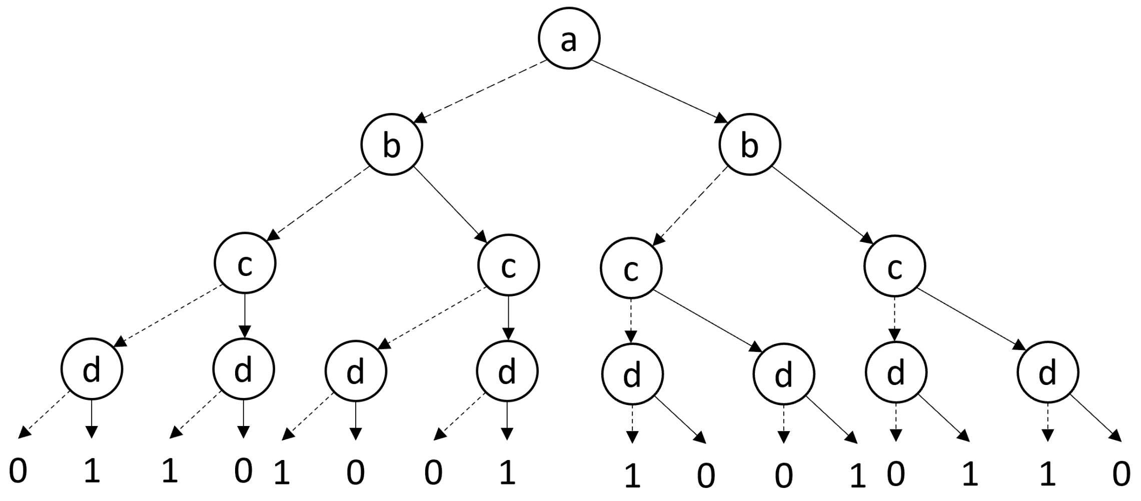


FIGURE 2.6: Binary Decision Tree generated from Table 2.2

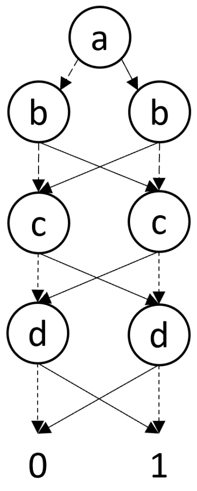


FIGURE 2.7: Binary Decision Diagram

transition relation T. BDD image computation starts with current reachable states S_c, performs the breadth-first search algorithm to calculate the next states reachable from the current set of states via transition relations T, such as $S_n = \exists V T(V, V') \wedge S_c(V)$, and grows reachable states by adding the next states $S_c = S_c \vee S_n$.

Given the BDD of the design and a safety property in CTL formula AGp, the basic idea of BDD-based symbolic model checking is to use BDD image computation to iteratively grow reachable states from the initial state and check whether there exists a reachable state that does not satisfy property p. For example, along all paths in Figure 2.7, check there there exists a state that does not satisfy property p.

However, BDDs can become too large. For instance, BDD size can grow exponentially for arithmetic circuits such as multipliers [3]. Though optimal variable ordering might be used to reduce the size of BDDs, finding a suitable ordering is NP-complete or needs manual intervention. There is no efficient ordering for some formulas, such as multipliers. As a result, formally verifying multipliers is a challenge.

2.4.3 SAT and Bounded Model Checking

2.4.3.1 SAT

The Boolean satisfiability (SAT) problem is: given a propositional formula u, determine whether there exists a variable assignment that makes the formula u to be true. If yes, u is called satisfiable. Otherwise, u is unsatisfiable.

BDD-based model checking requires converting a Boolean formula to a BDD and describing all satisfying solutions. On the other hand, a SAT solver only needs to find a single satisfying assignment to the Boolean formula, which is more efficient [55]. SAT is more scalable than BDDs [13] when dealing with large industrial circuits [3, 72]. The SAT solvers seemed to be a better solution than BDDs in model checking, because SAT procedures also operate on Boolean expressions but do not suffer from the potential state space explosion of BDDs; hence SAT-based Bounded Model Checking (BMC) was developed [74].

In BMC, a SAT procedure is used to replace BDDs; and Boolean formulas are constructed to check satisfiablity for a given finite path of length k. For example, given a property p and a Kripke structure K, the basic idea of BMC is to construct a satisfiable formula, if there exists a path of K whose length is bounded by k, and this path violates the given property p. The formula is then passed to an efficient SAT solver [55].

Most SAT solvers use conjunctive normal form (CNF) to represent formulas [75]. A literal is either a variable or its negation. A clause is a conjunction of literals. A CNF formula is a conjunction of clauses. A CNF formula is satisfied if all clauses are satisfied. One application is to encode all logic gates in a design and corresponding formal properties as CNFs, and combine all these CNFs to generate the overall CNF. Then use an SAT solve to solve the overall CNF.

One algorithm used by SAT solvers is Davis–Putnam–Logemann–Loveland (DPLL) [76]. The DPLL algorithm essentially performs a branch-and-bound algorithm to explore all variable assignments, until it finds a satisfying variable assignment (if the formula is satisfiable) or the search is exhausted.

2.4.3.2 BMC

In BMC, the model (state-transition system) is re-encoded using propositional variables. For example, a Kripke Structure is a (finite) set of states S, a set of initial states $I \subseteq S$, and a transition relation $T \subseteq S \times S$. The Kripke structure can be re-encoded using propositional variables only, and the result is purely propositional predicates I and T. The propositional formula $I(s_i)$ is true if $s_i \in I$, and $T(s_i, s_{i+1})$ is true if $(s_i, s_{i+1}) \in T$.

In BMC, the property is also modelled using temporal logic. For instance, AGp. The problem of finding a counterexample for the property p can be given as a finite path of length k that ends with a state s that satisfies $\neg p$. The problem can be formulated as:

$\exists s_0, ..., s_k.\ \ I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg p(s_k).$

There are three conjuncts in the above formula. The first conjunct states that the state s_0 must be one of the initial states. The second conjunct is the characteristic function of the transition relation between time step i and $i+1$. The conjunction operator creates k replicas of the transition relation T. The final conjunct asserts that the state s_k satisfies $\neg p$. If the above formula is satisfied, a counterexample is found.

One feature of BMC is sacrificing completeness for quick bug-finding speed. BMC is originally an incomplete bug-finding method rather than a complete verification method. However, some techniques enable unbounded proofs for a given property [77]. We will introduce some well-known techniques in the following.

One intuitive technique is increasing the bound k to meet the completeness threshold, but it is too difficult to compute the smallest threshold [55].

2.4.3.3 k-induction

Another method that extends bounded model checking to fully prove properties is k-induction [78]. For a state-transition system (I is the initial state and T is transition relation), a property p is inductive if p holds in the initial state I, and p holds in all states reachable from states that satisfy p. The basic idea of k-induction can be divided into two steps:

1) Base Case: check there is no counterexample within length k such that

$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg p(s_k).$

2) Inductive Case: check no state reachable from a sequence of k-states (that satisfies p) can violate p.

$(\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge p(s_i)) \wedge \neg p(s_k).$

If the base case is satisfiable, a variable assignment that satisfies the base case can be found and can be used to construct a counterexample for p. If both the base and the inductive cases are unsatisfiable, then return TRUE, which provides a sufficient (but not necessary) condition to fully verify the property p.

2.4.3.4 IMC

Interpolant-based model checking (IMC) can also complement BMC [79]. Most above techniques, such as BDDs, compute reachability explicitly. A different idea is building approximations of reachable states and refining them iteratively. In brief, IMC constructs an over-approximation of the reachable states and terminates after finding an inductive invariant or a counterexample.

An interpolant for an unsatisfiable formula $A \wedge B$ is a formula I, where $A \rightarrow I$; $I \rightarrow B$ is unsatisfiable; and I refers only to the common variables of A and B. The interpolant I can be used as over-approximations of A.

The algorithm of IMC of a property $P(s)$ is shown in the following, where $Init$ is the initial state, $get\_interpolant$ is a function to compute Interpolants.

```
if check($Init \wedge T(s_0, s_1) \wedge (\neg P(s_0) \vee \neg P(s_1))$)
    return False
$R = Init, k = 2$
while True // approximate reachability
  $A := R \wedge T(s_0, s_1), B := \neg P(s_k) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1})$
  if check($A \wedge B$)
    if $R == Init$
      return False
    else
      $k++$
  else
    $I$ = get_interpolant()
    $R = R \vee I[1/0]$ // map symbols at 1 to symbols at 0
    if $\neg$check($R \wedge T(s_0, s_1) \wedge \neg R(s_1)$)
      return True
```

Even with approximate reachability, the biggest disadvantages of IMC are requiring unrolling and restarting every time k is incremented.

2.4.3.5 IC3

The final technique introduced here to complement BMC is IC3. IC3 is considered as the most efficient single-engine model checking technique for proving properties of bit-level models; it is also shown to be able to reach deep counterexamples [55]. Invariants generated with the IC3 algorithm are known as property directed reachability (PDR) [80].

An inductive assertion for a transition system is a formula F which satisfies $I \Rightarrow F$ (called initiation and ensures all initial states are covered by F), and $F \wedge T \Rightarrow F'$ (called consecution and states that F is closed under the transition relation). F is inductive if both conditions are met.

A Counterexample To Induction (CTI) is a state which can reach a bad state in one step.

The formula F is an inductive strengthening of a safety property P if $F \wedge P$ is inductive.

The formula F is inductive relative to another formula G if both $I \Rightarrow F$ and $G \wedge T \wedge F \Rightarrow F'$ hold.

Similarly, P is an inductive invariant for the transition system described by T and initial states I if and only if $I \Rightarrow P$ and $P \wedge T \Rightarrow P'$ [16].

The basic idea of using IC3 to prove P is generating an inductive strengthening F of P covering all initial states [81]. Details are:

1) Check unsatisfiability of $(I \Rightarrow P)$ or $(I \wedge T \Rightarrow P)$. This is to ensure all initial states have to satisfy P and may not reach a bad state ($\neg P$) in one step. If either is unsatisfiable, return False.

2) Compute over/under-approximations of forward/backward reachable states. For example, build a sequence of frames F in CNF. $F[k]$ is an over-approximation of the states reachable in k steps.

3) Refines approximations by adding restrictions on states reachable in one step backward from a goal (bad) state. For example, in the ith step, proof obligation is (s, i) (s states).

IC3 handles this proof obligation by checking $F[i-1] \wedge \neg s \wedge T \Rightarrow s'$.

If unsatisfiable, $\neg s$ is inductive relative to $F[i-1]$, which means it is not reachable in one-step from $F[i-1]$.

Otherwise, a CTI is found. c is a sub clause of s. $\exists c.F[i-1] \wedge \neg s \wedge c \wedge T \Rightarrow s'$.

Which means there is a state contained in $F[i-1]$ that reaches s' in one step. Then add proof obligation (c, i-1) and recurse it.

4) Terminate when an inductive invariant or a counterexample (such as a bad state is proved to be reachable) has been found. For example, if for some i, $F[i]$ is inductive, then the property is TRUE; if the proof obligation is pushed to initial states I, then the property is FALSE.

There is no such algorithm that is always better than the others, though in general, BMC is good at unsafe checking (bug finding), IMC and IC3/PDR are better for safe checking (correctness proof) [82].

2.5 Related Work of Formal Verification

Formal verification has been widely used in academia and industry. In this section, we first review a wide range of formal verification, then we mainly focus on reviews that are highly related to our research. Detailed discussions of the highly related work, such as inspiration, limitation and motivation, will be given at the end of each review.

Due to the advantages of formal verification compared to traditional simulation, as stated in Section 2.4, formal verification can be used to validate the design across design stages. For example, Gao used formal verification to validate a CHERI-enabled processor [83]. Sharafinejad performed equivalence checking to verify the low-power processor designs that contain several power domains and different system-level power strategies [84]. Rojas combined both formal model checking and simulation to verify the functionality of a RISC-V processor [85]. Selvakumar used both equivalence checking and model checking to verify a Pseudo Random Number Generator [86]. Jakobs used both equivalence checking and model checking to verify processors with custom instruction set extensions [87]. Duan formally verified the connectivity and function of the I/O multiplexing in SoCs [88]. Drechsler applied BDD-based model checking to verify multiple types of adders [9]. Liew used SAT-based theorem proving to verify multiplexers [89]. Xiang and Duran used formal verification to identify bugs caused by misunderstanding of the ISA [90, 91]. Fadiheh applied formal verification to detect and locate vulnerabilities in covert channels [92]. Cheang introduced a formal methodology for enabling secure speculative execution on modern processors [93].

Fault analysis is a crucial aspect of achieving compliance with ISO26262. It involves identifying potential faults in the system, analyzing their effects, and ensuring that appropriate safety mechanisms are in place to mitigate risks. Many papers propose using formal verification approaches to perform fault analysis. We review some recent work that applied formal approaches to fault identification and classification in this section.

Jayakumar presented a methodology called model-based fault injection which can guarantee near-exhaustive state, input, and fault space coverage by applying formal

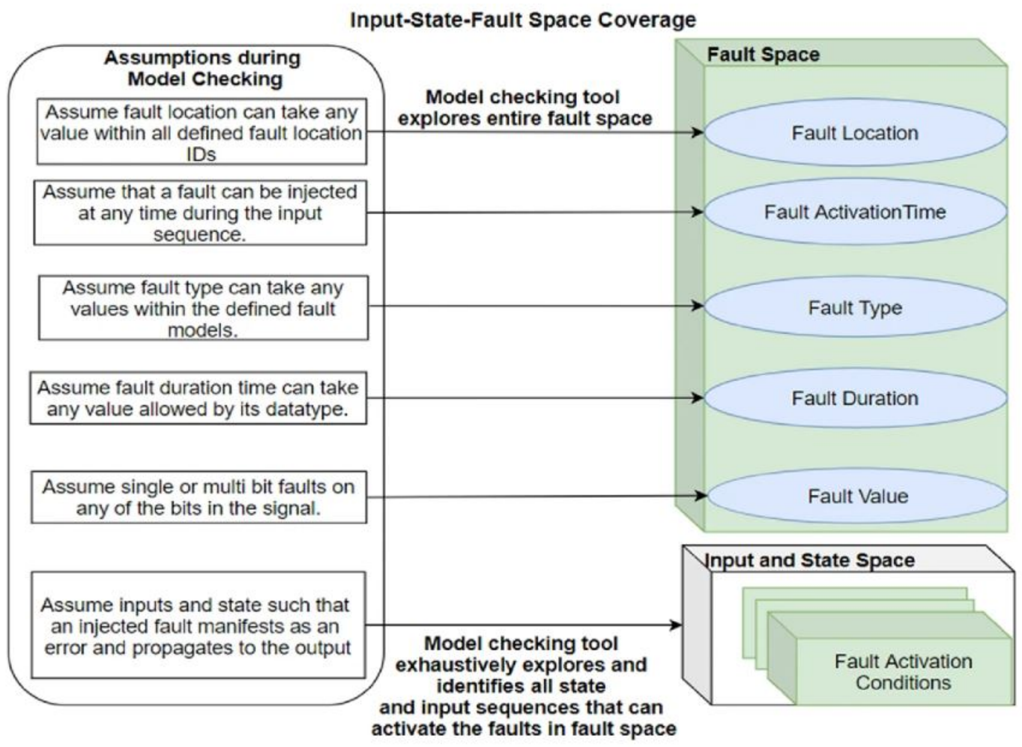

Input-State-Fault Space Coverage
Assumptions during Model Checking
Assume fault location can take any value within all defined fault location IDs
Assume that a fault can be injected at any time during the input sequence.
Assume fault type can take any values within the defined fault models.
Assume fault duration time can take any value allowed by its datatype.
Assume single or multi bit faults on any of the bits in the signal.
Assume inputs and state such that an injected fault manifests as an error and propagates to the output
Model checking tool explores entire fault space
Model checking tool exhaustively explores and identifies all state and input sequences that can activate the faults in fault space
Fault Space
Fault Location
Fault ActivationTime
Fault Type
Fault Duration
Fault Value
Input and State Space
Fault Activation Conditions

FIGURE 2.8: Assumptions to cover all faults [1].

verification [1]. This method is in fact a combination of fault injection and model checking. The method was implemented on Simulink behavioural models using the Simulink Design Verifier (SDV) model checking tool.

There are six steps in Jayakumar's method [1]. The first step is to implement fault injection mechanisms between signal drivers and receivers in the design, which will inject faults into the design after receiving fault control signals. The second step is to model fault propagation or error conditions as assumptions. There are a lot of faults in the fault list, considering the fault locations, injecting times, types, duration and values shown in Figure 2.8. Fault activation conditions specify the input/state sequences that cause the injected fault to manifest as errors and propagate within the system design. Manually listing the fault activation condition is tedious and time-consuming. Hence Jayakumar used the model checking tool to automatically list the fault activation conditions. To achieve this goal, the state/output conditions, which indicate an error or a failure, must be modelled as assumptions. These assumptions, as shown in Figure 2.8, will be used in later steps. The third step is to develop fault tolerance properties according to system safety requirements. Fault tolerance properties denote the expected system behaviour or primary outputs in the presence of faults.

The following three steps require SDV, the model checking tool [1]. The fourth step is to configure SDV to select supported fault locations and fault models from the assumptions in Figure 2.8. In addition, the model checker assumes that the fault can be injected at any time and for any duration, which allows a complete exploration of the entire fault list. The next step is to identify all the fault activation conditions for each fault chosen in the last step. In other words, use SDC to identify all the input sequences that can cause the injected fault to manifest as an error and propagate. The final step is to run a model checking of the safety properties. Fully proven results mean that the fault tolerance functionality can correct all the configured faults. Otherwise, a counterexample is generated to show the detailed fault attributes (location, duration, type, value and injection time) that bypass the fault-tolerance or safety feature.

This methodology successfully implements formal verification to cover all state space with faults. Jayakumar applied this methodology to verify safety properties by injecting a fault into the design. It is especially efficient for exhaustive single-fault injection. Compared to simulation-based fault injection, it saves orders of magnitude of execution time. However, this methodology suffers from a state space explosion for the Design Verifier model checker. Jayakumar gave no information about complexity control and formal techniques involved in his work. It is also not clear how Jayakumar handled inclusive results and what safety properties were developed.

Busch developed an automated formal verification flow to evaluate safety-critical registers in automotive micro-controllers [94]. Specification data of Special Function Registers (SFRs) in Extensible Markup Language (XML) format should be provided to the verification flow. For example, register level information such as register name, address, and reset value; and bit-field level information such as bit-field name and index range. Busch's method can be generally divided into three steps: 1) generate formal properties from the design model and specifications; 2) model check these properties (and other predefined properties); and 3) analyse the results.

Busch's work focused on functional verification of safety-critical registers [94]. Busch argued that the contents of these registers should not be corrupted by unintended software writes or spontaneous random bit-flips. Therefore he developed two groups of safety requirements: write access restrictions, and error detection & correction mechanisms. One example of access restrictions is that configuration registers can only be modified during the start-up phase. The second safety requirement is about redundancy-based error detection & correction mechanisms. A property was developed to raise an alarm by monitoring the difference between the original registers and duplicates. This property can cover all possible fault combinations in one single proof; iterative fault injection in each bit is not necessary.

To maximise the efficiency of this property, Busch developed a formal fault injection mechanism [94]. Given a simple register with an output port $q_o ut$ and an input port $q_i n$, ideally, they should be connected to keep the value in the register no change (suppose no write operation). The fault injection mechanism cuts the two signals into fan-in (q_out) and fan-out (q_in). Then assumptions were used to model fault models. For example, to constantly connect q_in to logic 0 to model stuck-at-0, and to connect q_out and q_in with an NOT gate to model a bit-flip.

The developed properties in Busch's work are divided into two types to reduce proof effort [94]. The first type is black-box properties. This type of property checks registers by monitoring bus interface signals, such as the write and read values of a register, hence bus protocol violations can also be checked. For example, checking if a register is modified by an illegal write, by reading this register before and after an illegal write access, and assuming there is no intermediate legal write to this register. Besides the above, the first type also checks legal writes, specific writes (such as mask writes and read-modify-writes), reset register value, and consistency of register values. The first type is suitable for verifying read-only or software-write-read-bit-fields-only registers. The first type of black-box property cannot cover all registers, such as registers with all bit-fields that are not read-accessible, or with additional internal write protections. Hence, the second type of white-box property is developed to perform similar checks on these registers. One difference between black-box properties and white-box properties is that white-box properties can assess the current state of a register instead of using an expensive read operation; thus it takes less time to prove white-box properties.

There are three things in Busch's work [94] that interest us. The first is what properties were generated to check safety-critical registers. Busch developed properties based on two safety requirements that safety-critical registers must hold. By proving these properties, faults that violate the safety requirements can be found. Though identifying all faults is not the main aim of Busch's work, Busch's work proved that it is practicable to explore faults using model checking. The second thing that interests us is the fault injection mechanism using cut points and assumptions. Though this mechanism supports various fault models, it cannot cover all bit-flips. It can only flip bits in the register output. If new data is written to the register at the same time the register output is flipped, which one will be stored in the register? This is a consideration in our research. The third thing we learn from Busch's work is using black-box to abstract some unnecessary parts can improve proof performance. We will utilize such a technique in this research.

Silva used formal verification to verify and prune fault lists generated by fault simulators tools in the context of safety verification [95]. Silva deployed Cadence Xcelium Fault Simulator (XFS) to generate fault lists and perform simulation-based fault injection. Then Cadence Functional Safety Verification (FSV) was used to perform fault analysis. Details of using FSV to perform fault analysis have been provided in Section 1.4.2. The IWLS 2005 benchmarks were chosen as case studies. Experimental results show that an average reduction of 29.5 % on the number of faults to be simulated is achieved. Silva's work has been integrated into the FSV. FSV is

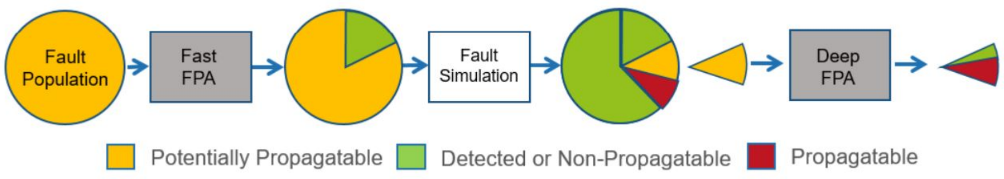


FIGURE 2.9: Integration of formal FPA with fault simulation. [2]

automated and fully integrated with XFS. Users can compile and elaborate the design, synthesize an Xcelium snapshot appropriate for the formal environment, load the snapshot into FSV, and run formal fault pre-qualification or sign-off. Integrating FSV before and after fault simulation reduces the fault set and eliminates useless re-simulation cycles. As mentioned in Section 1.4.2, FSV requires no formal language knowledge, as all required properties are automatically generated by the tool; users only need to specify fault lists and strobe points. Our research aims to develop and verify formal properties of SDCs, crashes and hangs. Compared to FPV, FSV can only partially meet our requirements because FSV cannot categorize faults based on SDCs, crashes and hangs.

Silva's idea and work [95] is similar to Marchese's idea and work [2]. Marchese proposed a two-mode formal fault propagation analysis (FPA) method that can integrate with simulation-based fault analysis to improve the functional safety of automotive SoCs [2]. The first mode, Fast Fault Propagation Analysis (FPA), targets large fault populations. Formal tools, which are configured to automatically select appropriate proof methods and strategies (such as proof engines in Cadence JasperGold), are used to identify the majority of safe faults. The second mode, named Deep FPA, requires more formal knowledge to analyse hard-to-prove faults. However, Marchese did not give details of the formal techniques involved in the two modes.

Figure 2.9 shows the integration of FPA with simulation-based fault analysis. The first step is to run FPA in fast mode to prune the fault list by removing safe faults that are detected or non-propagatable. Then perform simulation-based fault injection to test the majority of faults. There may be some faults that are hard-to-prove after simulation. FPA in deep mode can be used to further analyse and classify these hard-to-prove faults.

Based on the previous work [95], Silva combined Automatic Test Pattern Generators (ATPG), formal methods and simulation-based fault injection to decrease the efforts and improve efficiency of functional safety verification [96, 97]. Besides Cadence Xcelium and JasperGold which were used in the previous work [95], the Cadence Modus DFT Software Solution ATPG component is used to perform ATPG flow. The method begins with ATPG and Formal flows in parallel. After the ATPG flow is finished, the ATPG Testbench is used for simulation-based fault injection. Formal methods are applied to identify untestable faults and determine the behaviour of faults that are not covered by ATPG. Finally, at the end of all flows, the results from simulation and formal verification are retrieved and compared. For example, faults classified as Untestable by the simulation are equivalent to faults classified as Safe by formal verification and ignored by ATPG.

ISO26262 recommends fault injection to test reliability and safety. To reduce faults and to improve ISO26262 compliance, Silva proposed an approach to identify safe faults that do not disrupt safety-critical functionalities [98]. The method is applied to a CAN Controller IP, resulting in a Diagnostic Coverage of 93%. Silva argued that formal verification can identify more than Structural-Safe Faults, which are untestable by any valid test stimuli, and Detected faults, which are observable on safety-related outputs. Hence, Silva focused on Residual Faults, which can lead to system failures and are not covered by a safety mechanism. If such a fault cannot affect safety-related functionalities, then it is a Determined-Safe fault. Determined-Safe faults may affect the outputs of the design, but they cannot affect safety-critical functionalities.

Silva's method can be divided into two steps [98]. The first step is to run simulation to collect coverage data. Block and toggle coverage are the main focus. Block coverage determines whether specific states are activated in a state machine. Toggle coverage measures the activity of specific signals during the simulation. By analysing these two coverage reports, design elements that are not fully used during the simulation are Determined-Safe Candidates. These candidates will be further identified by formal methods.

The next step is to perform formal techniques, such as structural analysis, in FSV to find Structure-Safe faults. Formal verification generates COI for the target strobe point. Faults outside the COI cannot affect the target strobe point, hence are safe faults. Then the Determined-Safe Candidates generated from the last step are translated into formal rules, such as assumptions. The formal rules can increase the capacity of Safe Faults identification by limiting the input state space. FSV analysis is performed again with the formal rules to identify the Determined-Safe Faults.

Silva again used the similar method that relies on Cadence FSV and XFS to identify nodes that do not disrupt safety-critical functionality, enabling the reduction of undetected faults [99]. The method is almost the same as above hence not repeated.

Silva's method [98] extended the previous work [95] to identify safe faults that do not disrupt safety-critical functionalities. However, there are some limitations. The idea of using code coverage to prune the fault is not new. For example, dynamic slicing proposed by Volk [54]. Only stuck-at faults are evaluated. Many of the properties are not proven, which reduces the accuracy of results. Most of the experiments, including fault injection simulation using Cadence XFS and formal fault analysis using FSV, use automatic built-in tool features. Since FSV can exhaustively explore fault relations and

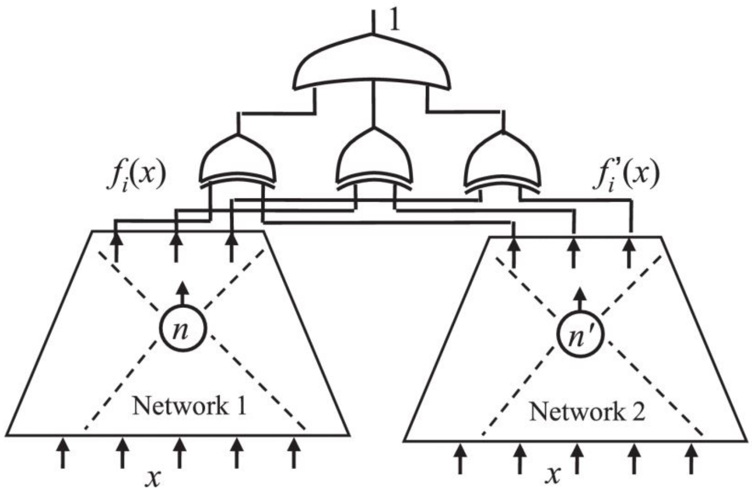


FIGURE 2.10: Equivalence checking miter for two copies of the network with different faults injected [3].

propagation, we argue that simulation-based fault injection is not necessary to identify safe faults.

In order to prune the fault list and speed up the functional safety verification process, Dao proposed a method that uses formal verification to identify equivalent faults [3]. Two or more faults are equivalent if the errors they produce at the outputs of the design are the same. Dao modelled a fault as a triple: a unique integer representing fault ID, the target node name and the type of the fault. Besides permanent stuck-at faults and negation faults, Dao also considered arbitrary functional changes, such as replacing a driver operator at a node with eight different logic gates, including (N)AND, (N)OR, (N)XOR, NOT and BUF.

As shown in Figure 2.10, Dao duplicated the design and added a miter circuit to prove the equivalence of two faults [3]. Dao modelled the design as a Boolean network and duplicated the network. Both networks are injected with different faults at nodes n and n'. The miter outputs 1 if any output of the two networks is different. Different faults are iteratively injected into each node in the networks.

The algorithm of Dao's method can be divided into three steps. The first step is to perform structural analysis to filter out nonequivalent faults [3]. Dao observed that if the primary outputs of two faults are different, or the two faults can affect different internal nodes / primary outputs, then these faults are likely not equivalent. The structural analysis results is fed into the second step. The second step uses both SAT and simulation to prove or disprove the true equivalence between any pair of candidate faults. The circuit in Figure 2.10 is transformed into a Boolean problem in the form of a CNF. CNF has been introduced in Section 2.4.3. An SAT solver is used the solve the problem, such as finding all assignments (inputs and faults) for which the miter output is 1. If unsatisfiable, the injected two faults are equivalent; if satisfiable, the faults are not equivalent; if undecided, the faults are assumed to be not equivalent. In the case of undecided results, case splitting is used for further analysis. The SAT solver can generate a test-pattern if two faults are not equivalent. In the third step, the test-pattern is fed into simulation to test all fault pairs in the same candidate fault class but with case splitting, i.e., splitting the class into several smaller candidate equivalence classes.

Dao's method is similar to other reviewed work that use both simulation and formal verification (such as equivalence checking) to perform fault relation analysis in order to prune the fault list. Dao used four benchmarks for demonstration. However, we would argue that the equivalent faults found by Dao are only equivalent under the four benchmarks (certain states) and are not general to all cases. In addition, Dao only focused on equivalent faults; Dao could further prune the fault list by performing fault propagation analysis to filter unpropagatable faults. What inspires us in Dao's work is using case splitting to handle inclusive formal results. Case splitting will be used in our research to handle undetermined results.

Huhn proposed an application-specific approach that applies formal verification to improve the robustness of sequential circuits [100, 101]. Huhn argued that the value of a flip-flop (FF) is often equal to the value of many other FFs at the same time. In addition, if the sequential circuit under test is known, it is possible to determine the relation between the FFs. For example, the states in which certain FFs can hold the same value. Under the above arguments, Huhn proposed to implement partial redundancy to protect FFs by comparing the values of equivalent FFs. Compared to other traditional space-based approaches (which introduce extra hardware to generate redundancy), timing-based approaches (which influence the timing behaviour) and application-specific approaches (which only consider the dedicated parts of a circuit under certain cases), Huhn claimed the proposed method can solve all the drawbacks, such as hardware overhead, latency and restrictions to general circuits. The most important part of the method is to correctly and exhaustively determine the relations between FFs for given applications, which is a computationally hard task. Huhn proposed the following Equivalence Property (*EP*). $P_j \subseteq N$ be a partition of at least two non-robust FFs N, and $\widehat{S} \subseteq S$ be a set of reachable states.

$$\mathrm{EP}\left(\widehat{S}, P_j\right) := \left\{ f_1, \ldots, f_l \in P_j \;\middle|\; \begin{array}{l} \text{all FFs } f_1, \ldots, f_l \text{ outputs the} \\ \text{same value under the same} \\ \text{state } s \in \widehat{S} \end{array} \right\}$$

EP should hold if all combinations of FFs $f_n, f_m \in P_j$ have the same output values in all reachable states $\widehat{S} \subseteq S$. Based on the above *EP*, Huhn can find all equivalent FFs in a partition and the states where all FFs in the partition hold the same value.

Specifically, Huhn used an SAT-based approach to solve the problem [100, 101]. The basic idea is that FFs of a partition P form a good partition if: 1) an assignment of the fan-in cone exists, such that an occurring transient fault in one $FF \in P$ is propagated and visible toward at least one Primary Output (PO); and 2) all FFs of P assume the same value. Let TP be a set of assignments of Primary Inputs (PIs), such that a transient fault of any FF of P is observable at the POs. All TPs are denoted as $TP(P)$. X defines the number of PIs as well as FFs in the fan-in cone of partition P. Then the TP metric (denoted as $M(P)$ for a given partition P) is:

$$M(P) = \frac{|\mathsf{TP}(P)|}{2^{|X|}}$$

The TP metric $M(P)$ can be used to evaluate and rank FFs.

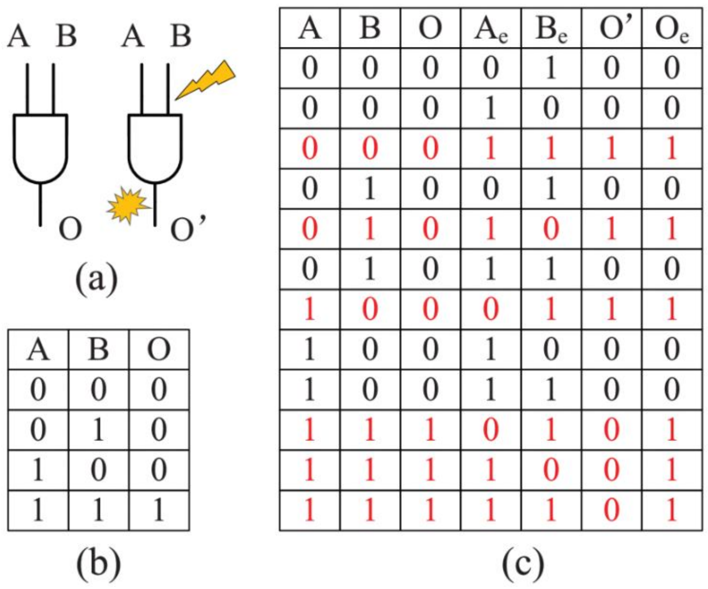
After finding the partition P_j where EP holds using SAT, the next step is to determine reachable states $\widehat{S}$ using BMC [100, 101]. Huhn formalised the problem into the following formula and used BMC to solve it. $\mathcal{P}$ is a logical formula modelling $EP(s_l, P_j)$. I is the initial state. l is the proof depth.

$$\mathsf{SFind}(P_j, l) = I(s_0) \wedge \bigwedge_{0 \leq i < l} T(s_i, s_{i+1}) \wedge \mathcal{P}$$

Instead of implementing redundancy-based fault-tolerant technologies to enhance reliability, Huhn chose to detect faults by utilizing the relationship among equivalent FFs [100, 101]. SAT and BMC were used to identify the equivalent FFs and the states where the equivalence holds. However, this method can only protect FFs under certain states - what about the other states where no equivalent FFs can be found? Another limitation is the choice of the proof depth in BMC. In Huhn's experiments, the proof depth l has been assigned to different numbers but without a clear explanation.

Hu developed formal fault effect propagation models which can be used for simulation, formal verification and emulation [4]. Three types of faults were modelled: bit-flip faults, stuck-at faults and random faults. The first two faults have been introduced before. An example of random faults is an X-state caused by uninitialized registers. If these registers are referenced before being configured with a meaningful value, the system could malfunction and observe unexpected design behaviour.

There are two steps to construct a fault effect propagation model [4]. The first step is to associate fault labels to all signal bits. A logic-high fault label indicates the labelled signal has a faulty value. The second step is to construct the model by duplicating the original circuit design and feeding faulty inputs to the faulty copy. The fault effects can be observed by comparing the outputs of the two circuit instances. For example, to model the propagation effects of bit-flip faults in an AND gate, $O = A \cdot B$, a bit-flip



A	B	O
0	0	0
0	1	0
1	0	0
1	1	1

A	B	O	A_e	B_e	O'	O_e
0	0	0	0	1	0	0
0	0	0	1	0	0	0
0	0	0	1	1	1	1
0	1	0	0	1	0	0
0	1	0	1	0	1	1
0	1	0	1	1	0	0
1	0	0	0	1	1	1
1	0	0	1	0	0	0
1	0	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	0	0	1
1	1	1	1	1	0	1

FIGURE 2.11: Propagation of bit-flip faults through a two-input AND (AND-2) gate. (a) An AND-2 gate. (b) The truth table of AND-2. (c) Partial truth table for modelling the propagation of bit-flip faults through AND-2. [4]

can occur at any input port, as shown in Figure 2.11(a). Figure 2.11(b) is the truth table of the original AND gate. Figure 2.11(c) shows the truth table with bit-flips, where A_e, B_e and O_e are the fault labels of A, B and O respectively. Based on the truth table, the Boolean formula of the fault effect propagation model for an AND gate is shown in the following, where O_e is the output in the presence of a fault:

$$O_e = A\overline{A_e}B_e + A_eB\overline{B_e} + \overline{A}A_e\overline{B}B_e + ABB_e$$

Hu's method [4] modelled faults and fault effect propagation in formal verification. Though Hu claimed his method could be used for formal verification, he did not perform verification to prove this claim. Only simulations were performed. Hence it remains a problem whether Hu's method can be used for formal verification. In addition, Hu's method was not automatic: the models were constructed manually. In fact, there are similar work about modelling fault propagation using formal methods. For example, Bagbaba represented SETs by multiple SEUs and performs formal approaches to analyse fault propagation; and experimental results showed that the fault space can be reduced by tens to hundreds of times [102]. However, these methods are only suitable for simple and small designs. It is hard to do similar work in a complex design. Hence, we do not use similar methods to model and analyse SEUs in our research.

Bernardini used formal techniques to simplify simulation-based fault injection and enhance the accuracy and comprehensiveness of functional safety analysis [16]. The basic idea is to find safe faults, prune the fault list by removing the safe faults, and then simulate the remaining faults in the fault list. The simulation-based fault injection is a well-known technique, hence we mainly focus on the formal part.

Bernardini performed model checking to find safe faults [16]. A fault will be safety critical if and only if a “bad state” can be reached as a consequence of the fault itself. Bernardini defined “safety” as a reachability problem and solved the problem with model checking, for example, proving a bad state can never be reached from initial states with known state transitions. In detail, IC3 algorithm-based PDR model checking (which has been introduced in Section 2.4.3) is used to find the invariant of the transition system under test. The invariant shows that the system remains safe with certain injected faults. It is unnecessary to simulate these safe faults.

The results show that Bernardini’s work can reduce timeout simulations and undecided results, which overall improves the performance of simulation-based fault injection [16]. We learn that it is theoretically and practically possible to combine fault injection with model checking. However, Bernardini’s work still focuses on simulation and cannot solve the problems of simulation-based fault injection, for example, not being exhaustive. ’Hardware model checking’ from Hardware Model Checking Competition is chosen as a benchmark for both model checking and simulation [103], which cannot cover all the input state space. Nevertheless, there are still inclusive/undetermined results, such as undecided faults. We think a possible improvement to Bernardini’s work is to use various approaches to further reduce the undetermined results.

Traskov used formal verification to perform fault analysis of a well-known safety mechanism, Triple Modular Redundancy (TMR) [104]. The generic flow for fault analysis includes fault population reduction, fault injection, checking and classification, and collection of metrics. Traskov thought the purpose of the safety mechanism was to detect and correct faults. Based on this, Traskov developed two aspects that must be verified. The first one is functional behaviour: the safety mechanism’s specification and underlying requirements must be satisfied. For example, to verify that the design behaves as expected without faults, the design behaves as expected with specific (’corrected’) faults, and the design goes into a safe state for a specified list of (’detected’) faults. The second one is diagnostic coverage: the safety mechanism is able to detect and handle enough faults.

Traskov combined fault injection with Sequential Equivalence Checking (SEC). SEC exhaustively compares both the logic and temporal behaviour of two given circuits. SEC has been introduced in Section 2.4. Traskov duplicated the design at gate-level: one golden as a reference and one faulty with fault injection; both designs were protected by TMR. A fault injector was used to control and inject faults to the faulty one. Three fault models were considered: permanent faults (stuck-at-0, stuck-at-1), transient faults (such as SEU), and intermittent faults.

Traskov used formal techniques to model faults and fault injection. The basic idea is to: 1) cut the destination signal *dest_sig* so *dest_sig* will be considered as a free net for the formal engines; 2) drive *dest_sig* with assumptions. The fault injection is controlled by a signal *control_sig*. If low, $dest_sig$ is driven by the golden copy *ref_sig*; otherwise, *dest_sig* is driven by the faults. For example, to model stuck-at-0 faults, the assumptions are:

```
assume property (control_sig |-> dest_sig==0 );
assume property (!control_sig |-> dest_sig==ref_sig );
```

Then formal properties in SVA are developed to check propagation and detection of faults. Three property examples are given. The first example checks whether an injected fault can (eventually) be observed at TMR, by comparing the inputs to the TMR in both copies:

$fault\_inject\_at(sig)|- > \#\#[0:\$] \bigcup_i (impl.inst.in_i == spec.inst.in_i)$

impl means implementation (the faulty copy), *spec* means specification (the golden copy). *inst* is the instance protected by TMR. $in_1, ..., in_i$ are inputs to TMR.

The second property verifies that for a particular fault, TMR (eventually) forces the system to go to a safe state.

$fault\_inject\_at(sig)|- > \#\#[0:\$](impl.inst.state == safe\_state\_value)$

impl.inst.state is the system state variable and *safe_state_value* is the system state safe value.

The third property models that a detected fault by TMR can also be corrected by TMR. In other words, the outputs of the two copies are the same.

$\bigcap_i fault\_inject\_at(sig) - > (impl.inst.out_i == spec.inst.out_i)$

Mentor's Questa Formal tool was used to perform SEC. The formal results and runtime were compared with those obtained using dynamic simulation-based fault injection techniques. Mentor's QuestaSim was used to perform simulations. Results showed that all faults in the TMR-protected elements can be detected and corrected. However, TMR added an extra majority voter; faults in the majority voter could propagate and cause errors.

There are some interesting things in Traskov's work that can be used in our work. The first thing is that Traskov developed constraints preventing multiple fault toggles from happening. Without such constraints, the fault control signal can toggle more than one time, and multiple faults will be injected. This information is important when exploring SEUs. The second thing we can learn from Traskov's work is implicit fault injection. Traskov did not force the fault control signal to toggle at all, thus the formal tool can decide whether to inject faults or not. In fact, implicit fault injection has been used in other work [105]. We will use implicit fault injection as a complexity control strategy, details will be given in Chapter 3.

Traskov [104] is not the only one to use equivalence checking to analyse faults and fault tolerance technologies. There have been many similar work using equivalence checking to check structural similarity and to evaluate fault tolerance technologies (mainly TMR). Berg verified whether three copies in TMR were equivalent and the voters were inserted as expected [106]. Beltrame used a similar approach but with fault injection [107]. Benites checked the equivalence of TMR-protected design with respect to the original design, and performed single fault injection to validate fault-tolerance of TMR [108].

Entrena used formal verification, mainly equivalence checking, to verify 1) fault tolerance techniques can prevent fault propagation by masking or detecting faults, and 2) the protected circuit is functionally equivalent to the original circuit without fault tolerance techniques [109]. Entrena's work outperforms the other work mentioned above [106, 107, 108], because it can effectively verify a wider range of fault tolerance techniques (more than TMR), such as Duplication with Comparison (DwC), Safe Finite State Machines and Hamming encoding. Figure 2.12 shows a flow chart of the overall fault-tolerance verification algorithm.

The fault-tolerant design and the original design are modelled in And-Inverter Graphs (AIGs). An AIG is a directed and acyclic graph that uses only AND gates and inverters to represent the Boolean functions of a circuit. Then functional reduction is performed to the fault-tolerant design to simplify its AIG. The reduction process includes using structural hashing for fast matching, performing simulation to identify equivalent nodes, and performing equivalence checking to find equivalent nodes. For example, in redundancy-based fault-tolerant circuits, combinational logic is functionally reduced. Then some external constraints, such as required values on particular nodes for safe FSM, are added to the AIG. Some fault-tolerant designs, such as TMR, do not require external constraints.

Entrena also formulated the research problem as a formal equivalence checking (EC) problem, and used implication-based Boolean reasoning to solve the problem. Boolean reasoning performs a systematic search for a consistent assignment on the circuit representation. It also facilitates the checking of error propagation constraints. Entrena extended circuit-based Boolean reasoning to sequential circuits with the help of timeframes. A sequential circuit can be unrolled in a set of timeframes connected by flip-flops. The flip-flop inputs are primary outputs of the previous timeframe, and the flip-flop outputs are primary inputs of the next timeframe.

Instead of checking two explicit copies of the circuit, Entrena used nine-valued models to represent the circuit [110]. In other words, each node in a circuit has two different

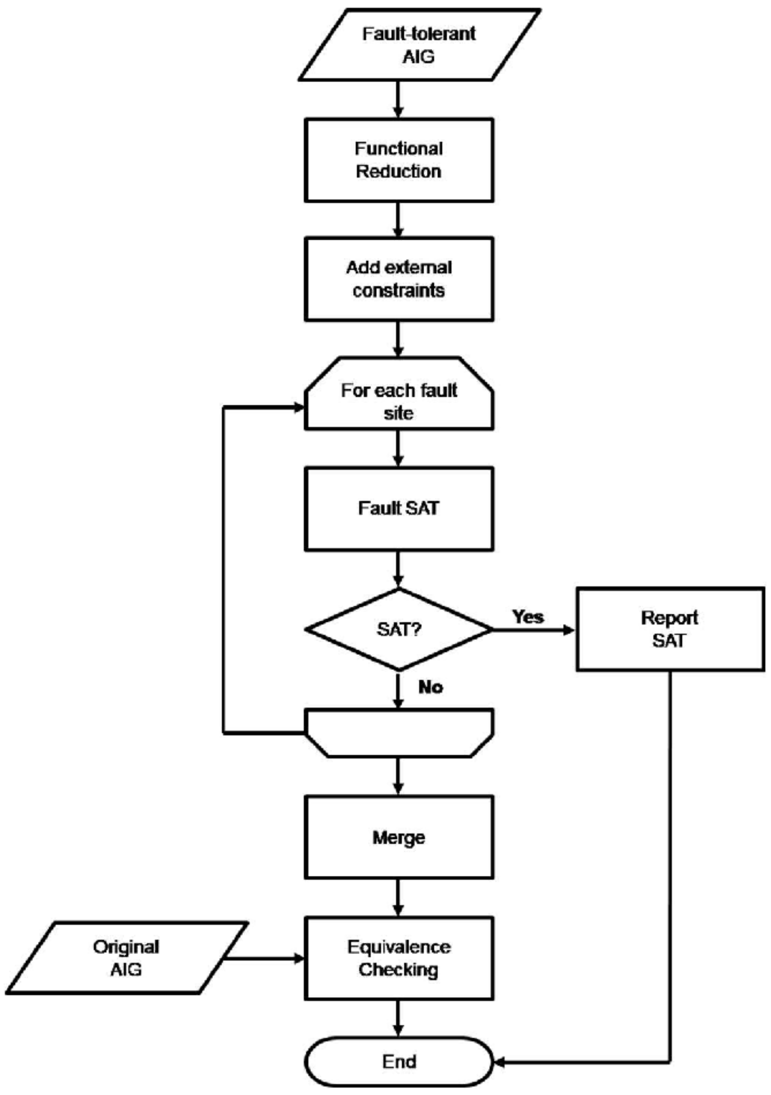


FIGURE 2.12: Integration of design and verification [109]

binary values: a good value v_g) and a faulty value v_f. Each binary value can be 0, 1 or X (unknown). Hence, the verification problem can be formulated as a satisfiability problem: given a target faulty node T, prove that the conditions $v_g(T) \neq v_f(T)$ and $v_g(O_i) \neq v_f(O_i)$ are not satisfiable, where O_i is a set of primary outputs.

The next step is to run fault SAT for each fault site. If the fault is satisfiable, report the satisfying test vector. If the fault is not satisfiable, merge redundant logic. For example, in a TMR-protected flip-flop, the three flip-flops are functionally equivalent and are merged after the fault SAT has proved no error can be propagated. At this point, most redundant elements (added by fault-tolerant technologies) have been removed, and the fault-tolerant circuit has a similar structure to that of the original circuit. Finally, equivalence checking is performed between the original and the fault-tolerant merged circuit.

The biggest difference between Entrena's work and other work lies in the equivalence checking step: Entrena performed equivalence checking in a hierarchical way without flattening the whole circuits. Entrena performed functional reduction to merge equivalent combinational logic and used fault SAT to merge equivalent storage elements, thus minimizing the size of the fault-tolerant circuit. This approach can reduce the computation efforts and thus would be scalable.

Krautz combined fault injection and formal verification to evaluate fault-tolerant designs [111]. Bit-flips are chosen as the fault model. Faults are injected into HDL by overwriting inverted values to target signals for one clock cycle. Based on the fault effects in the presence of fault-tolerance, Krautz categorizes the injected faults into 6 classes: 4 classes for the combination of error detected / not detected and error propagated / not propagated; and 2 classes for the behaviour error corrected / not corrected.

The quality of the fault-tolerant designs by evaluated by performing exhaustive fault injection on the hardened designs and measuring their fault coverage [111]. The verification process is similar to equivalence checking: Krautz duplicates the design (one golden and one faulty). The inputs to the two copies are the same and the outputs from the two copies are compared. A symbolical simulation engine based on BDD from [112] is used to perform the comparison (verification). BDD has been introduced in Section 2.4.2. Symbolical simulation constructs a BDD representation of the design. Then input patterns (faults) that satisfy the six fault classes can be found. The method essentially performs fault coverage analysis. The coverage results can be used to determine the error-detection and correction effectiveness of fault tolerance. The coverage is:

$$Coverage = \frac{DetectedandPropogatedFaults + DetectedbutNotPropagatedFaults}{TotalFaults - NotDetectedbutPropagatedFaults}$$

Krautz used a floating point unit (FPU) which was protected by Berger-Code-Prediction (BCP) for demonstration, and achieved an average coverage of 98.75% for the single bit error detection in about 42 hours [111].

Compared to other work that use equivalence checking to identify and classify faults, Krautz further classified single faults into six groups [111]. However, Krautz made an assumption that the fault effects must appear at the output in a fixed amount of time, and only performs verification for limited clock cycles. Little detail of the bound was given. Using fault coverage to analyse the effectiveness of fault tolerance is the main thing we can learn from Krautz'work. However, we would argue that the results may not be accurate due to bounded verification.

Kaja used formal verification methods to ensure thorough and reliable fault detection and correction in safety-critical designs that were protected by fault-tolerant technologies [5]. Kaja developed a special fault injection mechanism. To introduce this

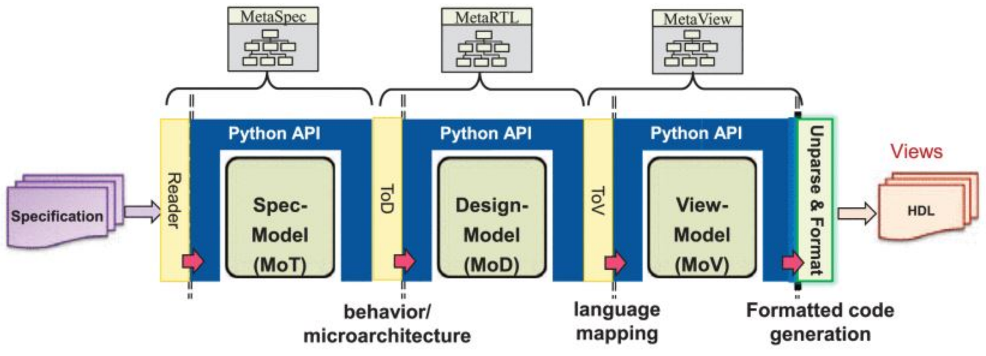


FIGURE 2.13: RTL generation flow by MetaRTL [5].

special fault injection mechanism, MetaRTL, a model-based RTL generation framework developed by Schreiner must be briefly introduced at first [113].

MetaRTL utilizes metamodels (such as construction, development, and documentation) to formalize design requirements, eliminating ambiguities and imprecise specifications [113]. Figure 2.13 gives the overview of generating HDL from design specification. The Model-of-Things (MoT) represents an instance of the metamodel, describing design configurations. The Template-of-Design (ToD) is a template containing the information on how to build all possible design instances. The Models-of-Design (MoD) is the RTL model tailored to a certain input configuration or specification.

Kaja extended the above hardware generation flow to support fault injection, as shown in Figure 2.14. MetaRTL is used to generate the design RTL, then the RTL is synthesized to generate gate-level netlist. The design RTL, gate-level netlist and fault injection targets are fed into the ToD Generator to generate a ToD in a mixed granularity fashion. For example, if the fault injection target is the Execute stage of a pipelined processor. Then the generated ToD defines the Execute stage at gate-level, and leaves the remaining processor in RTL. Fault injectors (extra hardware such as those introduced in Section 2.3) are added to the design model MoD, which is further transformed into a transformed MoD (T-MoD). More specifically, each fault injection target component in the MoD is added by a fault injector. These added fault injectors are controlled by fault control signals, which are primary inputs of the design. Fault collapsing is performed to detect equivalent faults and prune the fault list. SEUs are chosen as the fault model. To prove the functionality of the modified design (with fault injectors), Kaja used OneSpin to perform formal equivalence checking and verify the modified design is identical to the original RTL design, when there is no fault injected into the the modified design.

Kaja used two exhaustive techniques, MetaProp and Complete-Symbolic State Quick Error Detection ($C - S^2QED$), during formal-based fault injection [5]. MetaProp, which was introduced by Devarajegowda, is a model-based framework similar to

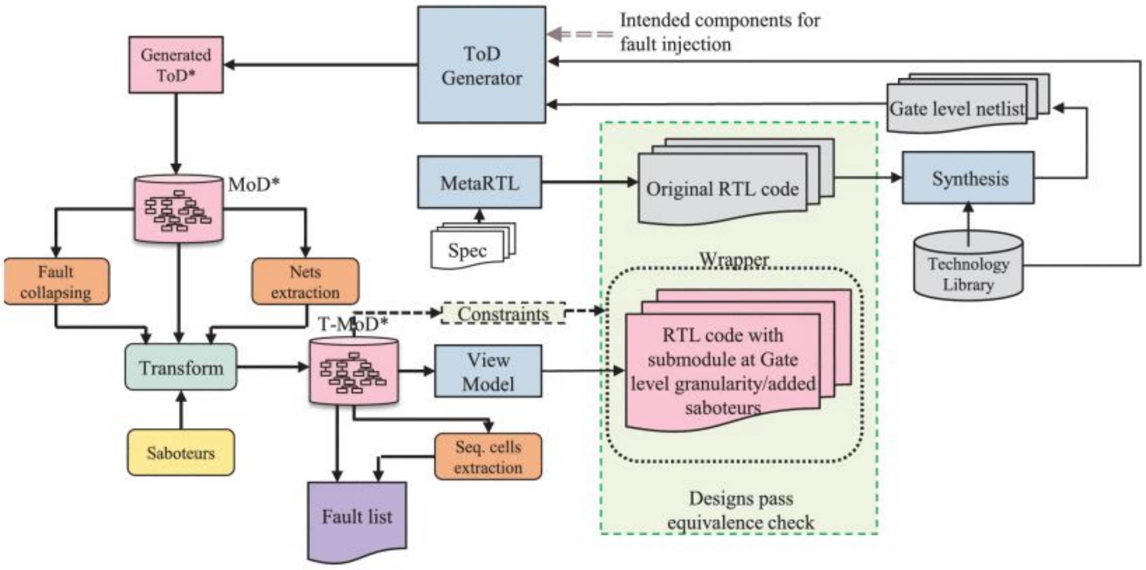


FIGURE 2.14: RTL generation flow by MetaRTL [5].

MetaRTL and can be used to automatically generate properties [114]. S^2QED, which was introduced by Fadiheh, is a BMC-based formal approach good at detecting instruction bugs resulting from a sequence of instructions [115]. The basic idea is similar to equivalence checking, for example, duplicating the CPU under verification. The two copies have the same input instructions, except that one CPU begins in a clean state (the pipeline is flushed); while the other CPU starts from a symbolic (free) state. S^2QED properties verify whether the two CPUs can remain QED-consistent (in other words, the register files of the two CPUs have identical contents). However, S^2QED fails to cover single instruction bugs. Hence Devarajegowda extended S^2QED to $C - S^2QED$, which covers both single and multiple instruction bugs [116]. Kaja further extended S^2QED to $FI - S^2QED$ by adding extra constraints (one CPU is fault-free while the other CPU is injected with faults) to S^2QED properties. Kaja used MetaProp to automatically generate these properties.

Kaja performed experiments on two 5-stage pipelined RISC-V processor cores, one with RV32IMC instruction sets and one with RV32IMCZicsr instruction sets [5]. TMR and ECC were implemented to harden the two processors. Three types of experiments were performed: the above formal method, a simulation-based analysis and an ATPG-based analysis. The results showed that the formal method achieved better results than the simulation-based analysis. The formal method also produced similar results to the ATPG-based analysis on combinational designs at the same time.

In Kaja's work [5], the first thing that interests us is the automatic fault injection mechanism generation flow. Kaja claimed that this fault injection flow and method makes formal methods scalable. However, we will not use the same MetaRTL-based method in our research, because we think it is too complex. We learn ideas about atomically implementing fault injection mechanisms into a design instead of copying the same work. Another thing that interests us is S^2QED properties. These properties duplicate the CPU and compare the Register File in two CPUs, and cannot detect single instruction bugs. We propose to model architectural behaviours of instructions as properties, which can detect single instruction bugs and require no design copy.

Liu proposed a fault injection and verification framework based on the Unified Modelling Language (UML) sequence diagrams (SDs) [117]. UML is a semi-formal language that can describe structures and behaviours of systems at different levels. However, UML has limited support for formal verification. Liu extended UML to formal verification by generating formal models from SDs, implementing fault injection rules to the formal models, and using a model checker to verify the formal models. TLA+ and TLC are used as the formal language and the tool [118]. The fundamental idea of combining fault injection and model checking is common: injecting faults into the formal model, using a model checker to search all the possible states automatically, and verifying if the system behaviour violates requirements that are specified as temporal logical formulas (formal properties).

The first step of Liu's method is to model the design as SD constructs [117]. An SD construct is a set of objects LL, MSG and the interactions ACT. LL (lifeline) represents either roles or objects that participate in the modelled sequence. MSG is the set of messages (or function calls, signals) enclosed in the SD, and defines communication between lifelines. ACT includes atomic actions (sending or receiving messages) directly enclosed in an SD and nested actions (joint atomic actions). Liu extended the SD constructs by adding variables, which describe properties, to the lifeline. The extended lifeline is $(name, atomACT, isActive, nvVAR, vVAR)$. $name$ is the lifeline's name. $atomACT$ are atomic actions (sending or receiving messages) along the lifeline. $isActive$ is the activeness of the lifeline. $nvVAR$ is a set of non-volatile variables; $vVAR$ is a set of volatile variables. The variable values can be changed by atomic actions or faults.

The second step is feeding the above SD constructs and properties to a tool to generate a TLA+ specification [117]. The TLA+ specification is similar to a combination of a transition system and temporal logic. The interactions in SD are interpreted as a sequence of states. The atomic actions (sending and receiving messages) are transformed into state transitions. All properties are transformed into temporal formulas. Liu models a fault in the form of $(type, location, condition, error)$. $type$ is the fault type introduced in the next paragraph. $location$ indicates the lifeline/action the fault is injected. $condition$ indicates the state the fault is injected. $error$ is the fault effect.

Liu categorized four types of faults [117]: (1) lifeline crashes ($isActive$ == False) and cannot do any actions, and its volatile variables $vVAR$ are reset to default values; (2) lifeline restarts from its initial state, and its volatile variables $vVAR$ are reset; (3) data corruption such as some variables of a lifeline are assigned abnormal values; (4) blocked message such as a message cannot be sent out or received. Liu implemented a fault injection mechanism by adding actions to TLA+ specifications. For example, once enabled, if $type$ is a lifeline crash, assign False to $isActive$ and initialize $vVAR$; for data corruption, assign abnormal values as defined by $error$ to $location$. Finally, the TLC model checker is used to verify the TLA+ model.

Liu's work focuses more on developing a formal verification tool/framework than just fault identification and classification. However, compared to other reviewed work that just care about safe faults and dangerous faults, Liu's work further classifies the dangerous faults into four groups. There are some limitations of Liu's work. The first limitation is the proposed framework has poor support to constraints or assumptions. The second limitation is poor support for multiple fault types, such as stuck-at-faults and bit-flips, since the framework is not sensitive to timing. Inspired by Liu's work and other simulation-based fault injection work, we want to use formal verification to further classify faults based on fault effects.

Fault tree analysis (FTA) is widely used to assess system reliability by identifying relationships between system failures and faults [119]. FTA starts from an undesirable event (top event) to find what circumstances may lead to the top event. The fault tree branches out vertically from the top event. Bottom events are occurrences that may lead to a component failure. Fault tree models are graphical representations of system failures, in terms of the system's components. Standard fault tree models consist of combinational logic (mainly AND and OR gates) and basic events. Traditional FTA cannot cover temporal behaviours. To solve the problem, Ammar proposed Temporal Dynamic Fault Trees (TDFTs) to capture temporal events (such as soft errors) [120]. Ammar introduced the sensitivity to Temporal Basic Events (TBE) to the fault tree gates by formalizing and modelling of the probabilistic behaviour of Fault Tree (FT) gates and events over time. Ammar developed the following probabilistic model (automaton) to represent a FT gate:

Given a TDFT gate with a set of inputs Y and an output Z, connected through certain logic (such as AND), the priced-timed automaton (PTA) of this gate can be formally defined by a tuple $A = (L, L_0, \chi, Act, P, \mathcal{L})$, where L is a finite number of states, L_0 is the initial state, χ is a finite set of clocks, Act is a finite set of actions over L, $inv : L \to \zeta(Y)$ is an invariant condition, P is a probabilistic transition function $L \times \zeta(Y) \times Dist(2^Y \times L)$, $\mathcal{L} : L \to 2^{AP}$ is a labelling function assigning atomic propositions to different states.

Based on the above definition, Ammar developed multiple probabilistic models to represent Temporal and Gate, Temporal or Gate, FDEP Gate, PAND Gate, and COMB Gate. To save space, the details of the construction process are not shown. Besides, it is out of the scope of our research.

Ammar's fault analysis flow starts from a system-level specification model provided by the designer in a general-purpose modelling language SysML. In the SysML model, the failure rate of each component must be characterized. Then a FT can be generated from any tool. Then add or replace original gates with TDFT gates developed by Ammar in order to get a formal PTA model of the system's FT. Then probabilistic model checking of the FTs is performed to evaluate the reliability of the system.

Similarly, Samadi used FTA to evaluate system reliability [121]. A key point in this work is to generate Dynamic Fault Tree (DFT) models based on statistical model checking (SMC). DFTs model the failure behaviour of the system via static and dynamic fault tree gates [122]. There are four steps in Samadi's method. In the first step, a library containing the models of all basic gates is created. The second step transforms the system under test to Fault Tree (FT) models according to the library in the first step. UPPAAL-SMC tool uses statistical model checking to reason about the formal specifications of stochastic systems. Samadi used this tool to model the probabilistic behaviour of the FT gates and the events as PTA formalism. Then the probability of system failure can be analysed from the FT model. It is not clear how to develop the library and how to construct the FI models. The last two steps are done by a tool. Details such as formal techniques used are not given.

Ammar's and Samadi's methods heavily rely on integrated tools. In addition, it is not clear what fault types are modelled and how they are modelled. Though FTA can be used to calculate the possibility of a system failure due to an erroneous component, component failure rates must be provided by designers [120] or calculated by integrated tools [121]. Hence it is not suitable for early design verification where the failure rates of each component are unclear. We think using FTA to identify and classify faults remains a challenge. It is proper to use model checking in our research.

Double Event Upsets (DEUs) have been a challenge in the reliability area. There are few works about formal verification in this area. Shao used probabilistic model checking to analyse the reliability of TMR and scrubbing with respect to Multiple Cell Upsets (MCUs) [123]. Shao combined Common Cause Failure (CCF) theory with a continuous-time Markov chain to model MCUs. Then Shao used a SAT solver for simulation. Frehse used formal verification to verify the robustness of a circuit with respect to multiple soft errors [124]. Frehse abstracted MEUs to prune the fault list and state space. Then Frehse modelled MEUs as SAT problems and used a SAT solver. Leveugle combined property checking and fault injection to identify unacceptable

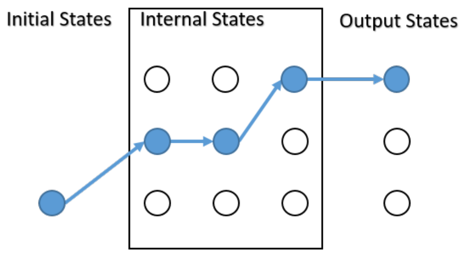


FIGURE 2.15: State path for one simulation

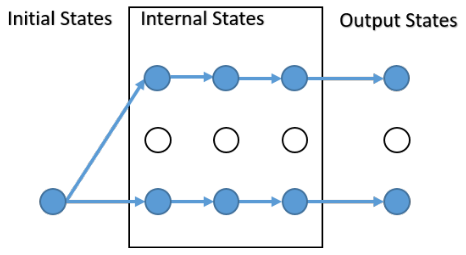


FIGURE 2.16: State path for multiple simulations

faults in early design stages [125]. Scalability to complex designs is a main issue of Leveugle's method [126, 127].

2.6 Comparison of Model Checking and Simulation

Model checking is different from simulation. Figures 2.15, 2.16, and 2.17 show state coverage of simulation and model checking. A hollow circle represents an uncovered state. A solid circle represents a covered state. In simulation, different input vectors can lead to different state paths. One simulation explores one state path. To reach and cover all state space, multiple simulations and carefully developed testbenches are needed. However, different inputs may lead to overlapping paths, which makes it hard for the simulation to reach some corner states. Model checking is independent of input vectors (requires no testbench). Model checking also does not depend on simulated clock cycles. From the current state, the next state can vary depending on inputs (different paths). All paths can be explored at once in model checking. Given an object state, model checking can automatically generate all necessary variables, including inputs, internal states, and clock cycles from the present state, to the object state. Simulation-based verification can achieve similar functions. However, simulation takes longer time and more effort. It is also hard for the simulation to list all solutions to the object state. Another benefit of model checking over simulation is that model checking is based on the present state. Which means the initial state can be arbitrary. The initial state of the simulation is the state after reset. To start a simulation from a special state, specific stimuli and time are required to reach the desired initial state.

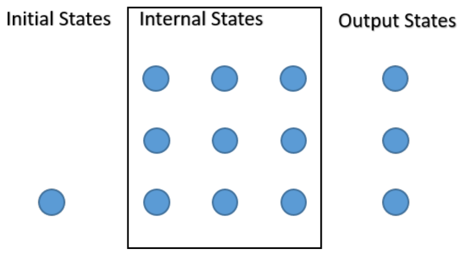


FIGURE 2.17: Model checking covers all states

In general, compared to simulation, the advantages of model checking include: 1) exhaustive analysis; 2) faster with higher state coverage; 3) independent of input vector, simulated clock cycles, and arbitrary initial state; 4) can list all combinations of inputs, internal states and time to reach the object state; 5) easy for debugging [55, 128, 129, 130, 131, 132, 133, 134, 135]. However, one major problem of model checking is state explosion: as the design size increases, the possible state combinations increase exponentially. For instance, there may exist 2^n states in an n-bit register. State explosion may be time-consuming. In addition, some complex properties are hard to prove by model checking. As a result, designs with a large COI may not be suitable for formal verification.

Apart from the above advantages, we chose formal verification because some problems are more suitable for or can only be solved by formal verification rather than simulation. One example is design verification. However, this research focuses on reliability. There are two scenarios where formal verification gives more benefits than simulation in the reliability area:

1) Formal verification produces higher fault coverage than simulation within the same verification time. Simulation-based fault injection is widely used to improve safety and reliability, as mandated by ISO 26262 for automotive applications [34]. To cover the entire state space and to explore all possible faults, multiple simulations are required. Considering the fault location and injection time, it is theoretically possible, but practically impossible, to simulate all faults for complex designs. Instead, random fault injection is used, which means a lower fault coverage. On the other hand, using formal verification to explore faults exhaustively is practically possible compared to exhaustive simulations. Inputs and initial states in formal verification can be arbitrary, which saves simulation time to the target states compared to simulation requiring a long input sequence to reach a specific state to test faults. In addition, some faults identified by formal verification cannot be identified by simulation [128, 129]. In this research, we need to perform exhaustive fault exploration, hence we choose formal verification.

2) Formal verification can perform root cause analysis of faults. Simulation is good at isolating faults and providing insight behaviour during runtime, hence it is suitable

for diagnosing operational issues. On the other hand, formal verification can isolate faults precisely at early design stages and requires no inputs or testbench, facilitating in-depth (interrelated) fault root cause analysis. Simulation injects faults and monitors responses, which is a forward tracing of faults. Given an error or a system failure, it is hard to use simulation to find the root fault. Formal verification can be used to perform backward tracing of faults: from the wrong results to identify root cause faults. For example, a crash is monitored and a corresponding assertion failure is reported. The root cause fault can be extracted from the corresponding counterexample. This research aims to perform root cause analysis of faults, hence we choose formal verification.

Formal verification may be computationally intensive. However, we would argue that a similar problem exists in simulation, as it requires extensive testing with diverse scenarios (especially for complex designs). Simulation is good at detecting operational faults that occur during system execution. On the other hand, formal verification can detect both functional and non-functional faults, including design errors, logical errors, safety violations, and temporal issues [65].

2.7 Summary

This chapter reviews some terminology and well-known verification technologies as well as their related work. In addition, the advantages, disadvantages, and suitable applications of simulation-based fault injection and formal verification are compared.

There have been many works that are related to our research, i.e., many works use formal verification to identify and classify faults. In general, most of the reviewed works use formal verification approaches as a support to simulation-based fault injection. In other words, formal approaches are used to prune the fault list, such as filtering structure-safe faults from the fault list, and then simulation is used to test the remaining faults. Only a few works use formal approaches to perform fault analysis. Most of the reviewed works use equivalence checking, which duplicates the design and compares outputs, to identify faults. Most of the reviewed works are good at identifying faults but poor at classifying faults: faults are just classified as safe faults and dangerous faults. Few works further classify the dangerous faults. There are many repeated works that formally verify and evaluate redundancy-based fault-tolerant technologies such as TMR, but there is no work about other fault-tolerant technologies such as residue arithmetic. There is also little work on DEUs.

We reviewed many related works and stated the influence of the reviewed work to us in this Chapter. Now we introduce the gap we want to fill. We want to use only formal verification approaches to perform fault analysis. We choose model checking in this research because it is more suitable to achieve our objectives. In the reliability area, there is a lack of methods to exhaustively explore all SEUs based on SEU effects in a reasonable time. We aim to identify and classify all faults based on the fault effects (SDCs, crashes and hangs) at the architectural level. Then we will expand the formal method to wider applications, such as evaluating fault-tolerant technologies and DEUs.

Chapter 3

Formal Method to Analyse SEUs

In this chapter, we use formal verification, mainly model checking, to perform fault analysis of SEUs: searching all candidate SEUs for the given SEU effects. Different from other works that use formal methods to identify safe and unsafe faults, we further identify and classify crucial SEUs into three groups: SDCs, crashes and hangs. We develop properties to reveal faults that lead to SDCs, crashes and hangs. To our knowledge, this is the first work of using formal verification to reveal faults that lead to three different types of failures.

The basic idea of our approach is: a) to construct the correct behaviour (based on the design specification) and fault effects (SDCs, crashes and hangs) as formal properties; and b) to use model checking to find all faults that may violate the corresponding property assertions. The properties are written as SystemVerilog Assertions (SVAs). Cadence JasperGold FPV, a commercial formal verification tool that supports Register Transfer Level (RTL) models and SVA, is used to perform model checking. We perform experiments on two servers with 24 $Intel^{R} Xeon^{R}$ E5-2670 Processors at 2.60GHz. The proposed method can exhaustively search the whole state space to find all candidate SEUs that may cause SDCs, crashes, and hangs in a reasonable time. We demonstrate this method by assessing the vulnerability of all structures (bits, registers and modules) in a RISC-V Ibex Core.

In model checking, there are three possible results of testing a property assertion: Proven, Undetermined (Bounded Proven), and Failure. If the status of a property assertion is 'Proven', the property assertion is fully proved. Faults in the corresponding bits cannot cause errors that violate the property assertion. An SEU is therefore deemed safe. As a result, these bits are not vulnerable to SEUs. If the status of a property assertion is 'Failure', the model checker generates a counterexample for the property assertion. SEUs in the corresponding bits may cause errors that violate the property assertion. The SEUs are therefore called crucial SEUs. As a result, these bits are vulnerable to SEUs.

State explosion, which is an inherent problem in model checking, is a major cause of 'Undetermined' results. If the status of a property is 'Bounded Proven', the formal tool cannot prove or disprove the property within a bounded trace or time. If there exists a counterexample, it is beyond the maximum trace or time limit. In other words, it is hard to find a fault that can fail the property assertion. The bit is undetermined with respect to SEUs. Section 2.4 reviews some formal technologies to extend BMC to unbounded results. We aim to utilize those technologies and develop various complexity control strategies to solve all the undetermined results in this research.

3.1 Method Overview

Figure 3.1 shows the overview of the proposed method to identify and categorize SEUs. The method can be divided into four steps, as labelled from ① to ④ in Figure 3.1. We briefly introduce the workflow of the four steps in this section.

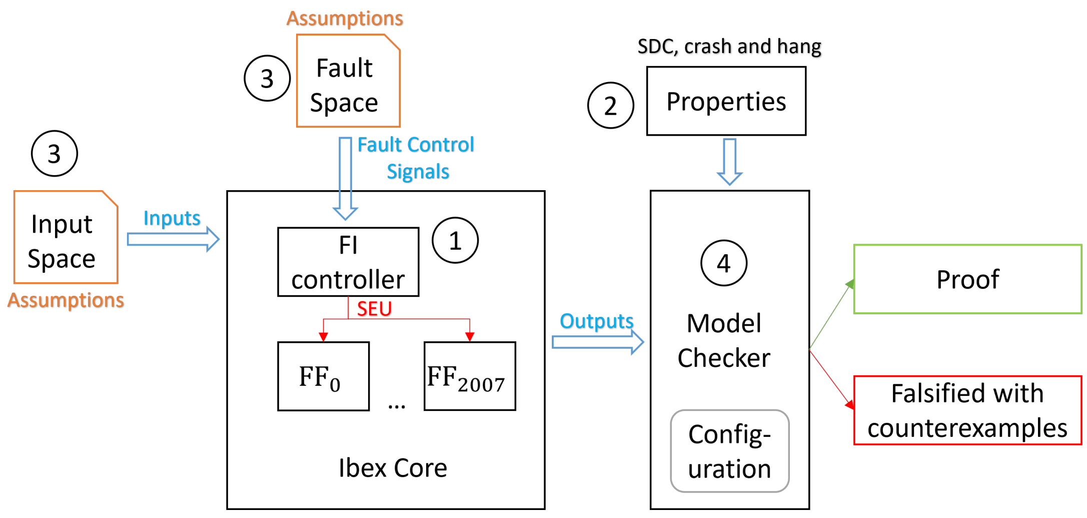


FIGURE 3.1: Overview of the proposed method to analyse SEUs

The first step is to implement a fault injection mechanism into the Ibex Core. The fault controller in Figure 3.1 reads fault control signals as primary inputs, and injects an arbitrary SEU to the Ibex Core. The details will be given in Section 3.2.

The second step is to model failures caused by SEUs as safety properties. We developed three types of properties to identify and classify SEUs that can cause SDCs, crashes and hangs. The details of the developed properties are in Section 3.3.

The third step is to develop various complexity control strategies to improve proof performance and handle state explosion. The complexity control strategies are more than the SVA assumptions in Figure 3.1. The details are in Section 3.4.

The final step is to configure the formal tool and perform model checking. The model checker proves or disproves the developed properties in the presence of SEUs. Different configurations are used to solve different problems. The details of the formal tool settings and the experimental strategy are in Section 3.5.

To ensure the correctness of the proposed method, each step and the whole framework are verified and validated using various approaches in Sections 3.6 and 3.7.

3.2 Step 1: Fault Injection

3.2.1 Fault Model

We choose SEUs as the fault model. There are three attributes of the fault model: the location where a fault occurs, the time when the fault occurs, and the period for which the fault exists. We assume that each bit-flip lasts until the bit-flip is refreshed by the processor. As a result, we do not control the fault period explicitly. We only model the first two attributes as the fault control signals. We are working at Register Transfer Level (RTL). The physical proximity of bits is out of consideration; all bits are treated equally. In other words, we assume SEUs could occur in all bits equally, irrespective of locations.

3.2.2 Fault Injection Mechanisms

The first step of our method is to implement a fault injection mechanism. We need to model faults during formal verification in order to explore fault effects. Some formal tools have such functions. However, Cadence JasperGold FPV does not have this function. More importantly, we aim to develop a general method that can be used in other formal tools. There are three general approaches to injecting faults.

The first approach is to use *force* and *release* statements in SystemVerilog. The IEEE Standard 1364-2005 defines the *force* and *release* procedural statements [136]. The *force* statement overrides all procedural assignments to a variable or net. The *release* statement ends a procedural continuous assignment to a variable or net. The value of the variable remains the same until it is assigned a new value. The following is an example:

```
task seu_force_net;
//This task overrides drivers of the bits in the LHS with a
//bit-flip until a release statement is executed
  input bit_ID;
  integer bit_ID;
  begin
    case (bit_ID)
       0 : force DUT.reg_1.bit_1 = ~DUT.reg_1.bit_1;
       1 : force DUT.reg_1.bit_2 = ~DUT.reg_1.bit_2;
       ......
    endcase
  end
endtask

task seu_release_net;
  input bit_ID;
  integer bit_ID;
  begin
    case (bit_ID)
       0 : release DUT.reg_1.bit_1;
       1 : release DUT.reg_1.bit_2;
       ......
    endcase
  end
endtask
```

However, this approach is not suitable for our method. The first reason is that Cadence FPV has poor support of *force* and *release* statements because they cannot be synthesized. The second reason is that it is hard to flip correct data from combinational logic instead of a register itself. Below is an example of a register in SystemVerilog. The register stores input data if enabled. If the *force* statement is executed at the same time the register is enabled, the drive of the register is cut (the input data is lost), and a bit-flip occurs in the wrong value. This may lead to failures not caused by SEUs, hence a false negative result.

```
always_ff @(posedge clk_i or negedge rst_ni) begin
  if (!rst_ni) begin
    reg_1 <= '0;
  end else begin
    if (enable) begein
      reg_1 <= input_data;  end
  end
end
```

The second approach is by cut points and assumptions: disconnecting the target bit driver and developing an assumption to reverse the bit [94]. The following is an assumption example.

```
assume property (@ (posedge clk_i)
 if (fault_injection_time_meet)
   bit_1 <= ~bit_1 );
```

This approach is, however, also not suitable for our research. The first reason is that *if...else..* statement is used inside the property, which is not recommended by JasperGold, because of the increasing complexity of compilation and model checking. In addition, each property can only inject one bit-flip to one bit location; multiple properties are required to inject faults in all bit locations. In addition, if new data is written to the bit at the same time the bit output is flipped, a false negative occurs. Moreover, it is hard to control only a single fault - it is hard to enable only one fault injection property to be active per model checking run. Additional auxiliary (AUX) code is required, which increases both design and verification complexity, and may alter the behaviour of the design under test.

The final and most widely used approach is to implement a fault injection mechanism. There are multiple fault injection mechanisms: injecting faults using software or scripts [14, 43, 53], implementing extra hardware for fault injection [36]. In this research, faults are injected by adding extra gates into the SV model.

There are two components in our fault injection mechanism: a Fault Injection (FI) controller module and multiple XOR gates. Source codes of the FI controller and XOR gates are provided in Appendix D. Fault location and fault time are two important attributes in simulation-based fault injection. As shown in Figure 3.1, the FI controller reads the two fault control signals, fault location and the injection time, as primary inputs to the Ibex Core. The outputs of the FI controller module are mask signals to all XOR gates. The XOR gates are introduced in the next paragraph. There is a counter in the FI controller. When the counter reaches the injection time, the target XOR gate is triggered to inject a fault into the destination bit.

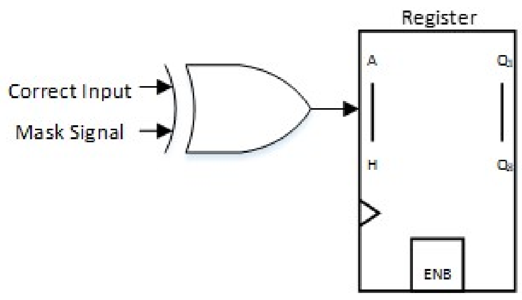


FIGURE 3.2: An XOR gate to inject faults

A two-input XOR gate is added before each register, as shown in Figure 3.2. The inputs of the XOR gate are the correct data to the register and a one-hot encoded mask signal from the FI controller. The correct data is driven by other logic or the output of the register itself. The mask signal determines the bit flip location of the original data. As shown in Table 3.1, a logic high in the mask signal can flip a bit in the corresponding position in the correct data. The output of the XOR gate is fed into the input port of the register.

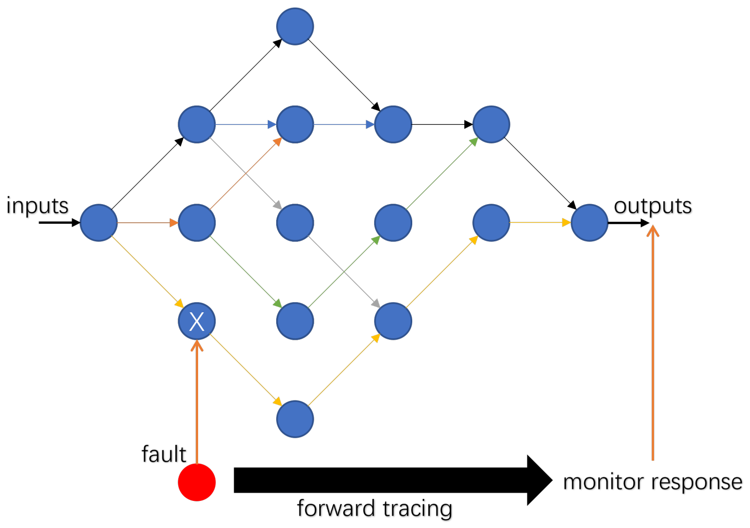
TABLE 3.1: Truth Table of XOR Gate

Mask Bit	**Correct Bit**	**Output**
0	0	0
0	1	1
1	0	1
1	1	0

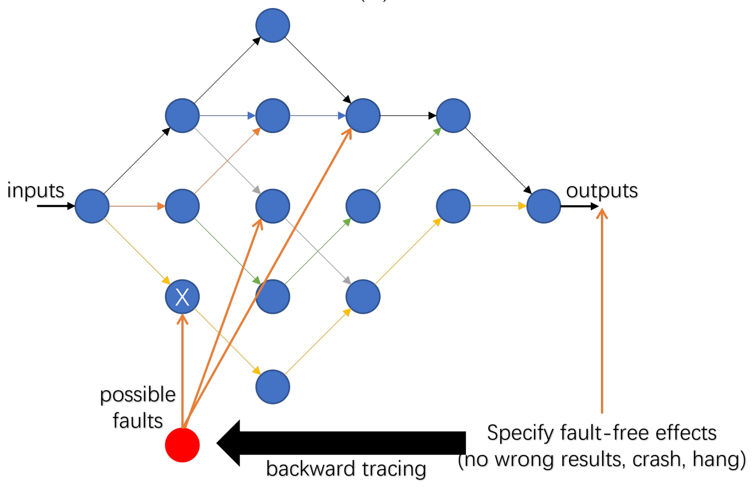
3.2.3 Implicit Fault Injection

We aim to perform backward tracing of faults. Injecting a fault and monitoring the response (such as simulations) is forward tracing, as shown in Figure 3.3a. A circle represents a state. Arrows through the state space are paths. Each run explores at most one fault and one path. In contrast, for backward tracing of faults, Figure 3.3b, we specify fault-free behaviours at the end; we do not specify fault attributes. Then we use model checking to find faults that violate the fault-free behaviours and cause errors and system failures. All paths can be explored at once.

As stated before, the fault location and the fault injection time are two primary inputs. Compared to simulations that control the fault location and the fault injection time explicitly, the biggest difference of our method is specifying the fault location and the fault injection time implicitly: we do not assign any values to the two inputs. In simulations, assigning no value to an input is a problem. Things are different in model checking. Inputs without certain values are treated as unconstrained inputs: the fault location and the fault injection time can be any values during model checking. The formal tool will explore all combinations of the fault location and the fault injection time to prove assertions. Hence an arbitrary fault can exist in the Ibex Core during model checking. The arbitrary fault can occur at any bit location at any time, hence the whole fault space is covered. The formal tool can explore the whole fault space to identify a fault that causes a failure (to find a counterexample that causes an assertion failure). The whole process implies backward tracing of faults, because faults are identified from the fault effects by proving assertions instead of injecting an explicit fault and monitoring response. There are multiple assertions. Multiple faults violating



(A)



(B)

FIGURE 3.3: (a)Forward tracing of faults using fault injection (b) Backward tracing in this method

different assertions can be identified from one model checking run, which is another advantage over simulations (that can identify at most one fault per simulation).

However, after experiments, we realized that multiple faults were injected per model checking run. This is caused by under-constrained assumptions, because we aim to explore SEUs rather than MEUs. The problem is solved by developing the following two assumptions. `FI_time` is the primary input to the Ibex Core that controls the fault injection time, `FI_index` is the primary input to the Ibex Core that controls the fault injection bit location. The following two assumptions make sure the fault injection time and the fault injection location do not change during model checking, hence we are injecting an SEU rather than MEUs. We use the range from 0 to 2007 to represent all the register bits in the Ibex Core. The bit index is shown in Appendix E. The second assumption also limits the fault injection range, because there are only 2008 bits in the

Ibex Core.

```
assume_FI_time_stable:
assume property ( @(posedge clk_i) $stable(FI_time) );

assume_FI_index_stable:
assume property ( @(posedge clk_i)
  $stable(FI_index)&&(FI_index<2008) );
```

Without the two assumptions, `FI_time` and `FI_index` can change at every clock cycle. The worst case is that a different fault is injected into the core at each clock cycle. For example, a fault is injected to bit_3 at the first clock cycle, then a fault is injected to bit_1 at the second clock cycle, and after that a fault is injected to bit_5 at the third clock cycle...

Besides the two assumptions, there is no assumption of `FI_time`, such as a time window during which a fault is assumed to occur. We assume a single fault can occur at any clock cycle after reset and can only occur once per model checking run. Similarly, we assume a single fault can occur at any bit after reset and can only be injected once per model checking run. However, during experiments we add extra assumptions about `FI_index` to reduce the fault space by assuming the identified vulnerable bits will be protected, the details are in Section 3.5.1.

In summary, we implement extra XOR gates to inject faults and develop an FI controller to control the XOR gates. We develop assumptions to ensure that only a single SEU can be injected per model checking run, and this SEU does not change during the model checking run. We implement implicit fault injection to inject an arbitrary SEU: we set the fault control signals as free nets so the model checker can cover the whole fault space. We validate the fault injection mechanism in Section 3.6.1.

3.3 Step 2: Formal Properties

The second step is to develop formal properties that can identify and classify SEUs according to the SEU effects. Section 2.1 categorizes SEUs into four groups based on the four types of SEU effects: no effect, SDCs, crashes, and hangs. Therefore, we aim to develop three groups of properties to find faults that can cause SDCs, crashes, and hangs. There is no need to develop a fourth group to cover no effect, because faults that cause no effect can also be identified by the other three groups. We introduce these properties and how they are developed in this section. The developed properties are validated in Section 3.6.2.

3.3.1 SDC

To develop properties that can identify crucial faults that may cause SDCs, the SDC features must be specified clearly: when an SDC occurs, the behaviour or the outputs of the design differ from the golden design. The behaviour can be compared at different levels. For example, the architectural level can monitor executing instructions, and the microarchitectural level can monitor state transitions. Based on the above features, two types of SDC properties are developed: architectural properties and strobe properties. In the following paragraphs, we demonstrate and compare these two properties.

3.3.1.1 Architectural Properties

Architectural properties specify the correct behaviour of each instruction. In this research, the RISC-V Instruction Set Manual is referenced to develop architectural properties [6]. We develop architectural properties that specify the architectural behaviour of a RISC-V core when executing an RV32IMC instruction. There are 40 RV32I instructions, 8 RV32M instructions, and 25 RV32C instructions. Each instruction has a unique architectural property.

We use the `BGE` instruction as an example to demonstrate how to develop an architectural property. Based on the RISC-V Instruction Set Manual [6], the `BGE` instruction uses the B-type instruction format, as shown in Figure 3.4. `imm` means immediate value. `rs1` and `rs2` are two source register addresses. `funct3` is the function code. `funct3` and `opcode` determine an instruction. The 12-bit B-immediate encodes signed offsets in multiples of 2 bytes. The offset is sign-extended and added to the address of the branch instruction to give the target address. The conditional branch range is ±4 kB. `BGE` instruction compares two registers. `BGE` takes the branch if the value in register `rs1` is greater than or equal to the value in register `rs2`.

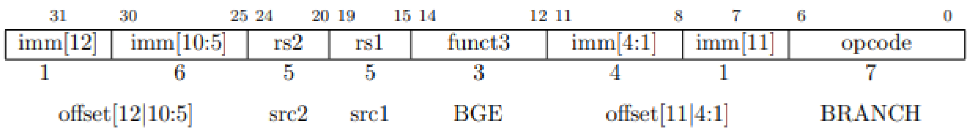


FIGURE 3.4: Format of `BGE` instruction adopted from [6]

To develop an architectural property about the `BGE` instruction, the first step is to specify the antecedent. The antecedent is a sequence (which can be wired to a signal) specifying if the executed instruction is the target instruction, in this case, a valid 32-bit `BGE` instruction. Figure 3.4 shows the `funct3` and `opcode` of `BGE`. The Ibex Core supports both 32-bit and 16-bit instructions. In addition, the instruction is wired from the processor and only valid if valid signal is high. It is necessary to check the validity and bit length of the retired instruction. As a result, the antecedent of the property is:

```
wire RV32I_BGE = valid&is_rv32_insn&
(funct3==3'b101)&(opcode==7'b1100011);
```

valid is set for one clock cycle after the processor completes executing an instruction. is_rv32_insn is true if the retired instruction is 32 bits.

The next step is to develop the consequent of the property. The consequent compares the branch address from the Ibex Core (which is the write data to the PC register) with the expected branch address (which is the theoretical write data to the PC register). The theoretical value can be calculated easily:

```
wire [31:0] next_pc_bge =
$signed(rs1_rdata)>=$signed(rs2_rdata)?
(pc_rdata+imm_b_type):(pc_rdata + 4);
```

rs1_rdata and rs2_rdata are signed compared values from two source registers. signed() function treats them as signed values. If the first value, rs1_rdata, is greater than or equal to the second, rs2_rdata, a branch will be taken. next_pc_bge is the theoretical value of the next program counter. pc_rdata is the current program counter value. imm_b_type is the immediate value extracted from the instruction:

```
assign imm_b_type = { {19{insn[31]}}, insn[31], insn[7],
                          insn[30:25], insn[11:8], 1'b0 };
```

In the end, the property is developed:

```
property p_RV32I_BGE;
  RV32I_BGE |-> pc_wdata == next_pc_bge;
endproperty
```

pc_wdata is the next program counter value extracted from the Ibex Core. next_pc_bge is the theoretical value of the next program counter.

The above architectural property specifies the correct behaviour of the BGE instruction. When there is no fault, the property assertion is proven without a vacuous pass. If the assertion fails, a fault altering the correct execution of the BGE instruction (such as a fault causing a wrong comparison result) is found.

However, during experiments we observed that the architectural properties cannot cover all SEUs, such as SEUs in the Register File and SEUs in the instruction immediate-field. The detailed results are in Section 3.8. As shown in the BGE example, the source register values rs1_rdata and rs2_rdata and the immediate value `imm_b_type` must be extracted from the Ibex Core to generate theoretical values. But what if the theoretical values are calculated based on the incorrect signals extracted from the Ibex Core?

On the one hand, the Ibex Core directly reads the source register values from the Register File and the immediate value from the fetched instruction; there is no safety mechanism to detect/correct faults in the Register File or the immediate-field. On the other hand, the signals used in the architectural properties are directly extracted from the Ibex Core, and there is no golden reference to determine whether these signals contain faults. As a result, architectural properties cannot cover faults in the Register File and the instruction immediate-field. To cover faults in these structures without implementing extra safety mechanisms, an extra design copy is required as the golden reference. Based on this observation, strobe properties are developed to improve the SDC fault coverage.

3.3.1.2 Strobe Properties

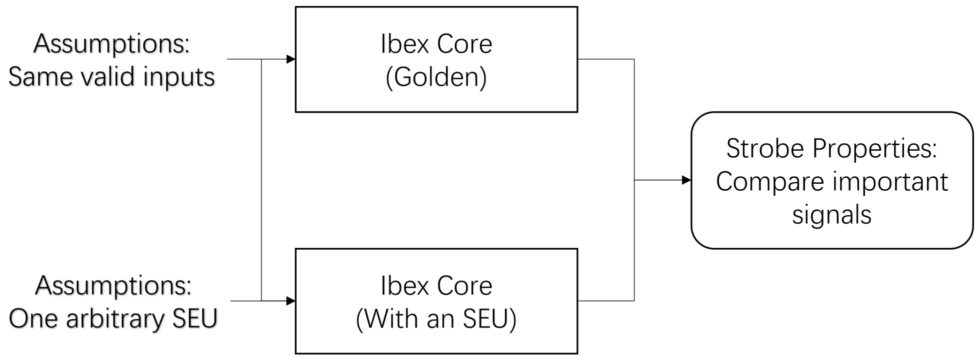


FIGURE 3.5: Diagram of Strobe Properties

The idea of using strobe properties to detect failures and identify faults is motivated by Equivalence Checking and Cadence FSV. The Ibex Core is duplicated: one golden core and one faulty core with an arbitrary SEU. Strobe properties compare important signals used in architectural properties (such as the instructions and the source register values) in the two Ibex Cores. There are 15 such signals, whose name and description are listed in Table 3.2. These strobe signals can be wired as outputs for comparison. As shown in Figure 3.5, input instructions and data to the two cores are the same, which is achieved by simply wiring the same inputs to the two cores. Faults in the structures that cannot be covered by architectural properties, such as Register File, can be identified easily by comparing the source register values in the two cores.

```
assert_valid: assert property (
 golden_valid == faulty_valid );
```

TABLE 3.2: Strobe Signals

Name	Description
valid	An instruction has been completed and retired from the core.
insn	The retired instruction
rs1_addr	Address of the source register 1
rs2_addr	Address of the source register 2
rs1_rdata	Read data from the source register 1
rs2_rdata	Read data from the source register 2
rd_addr	Address of the destination register
rd_wdata	Write data to the destination register
pc_rdata	The current Program Counter value
pc_wdata	The next Program Counter value
mem_addr	Memory address
mem_rdata	Read data from the memory
mem_rmask	Read mask to the memory read data
mem_wdata	Write data from the memory
mem_wmask	Write mask to the memory write data

The above is the first strobe property assertion we developed (clock and reset signals are not shown). When both cores finish executing a valid instruction, `golden_valid` and `faulty_valid` are set for one clock cycle. We developed this property at first because `*_valid` signals are the most important signals in strobe properties. If they are not equal, there is no doubt the two Ibex Cores behave differently. In addition, signals compared in other strobe properties are only valid when `*_valid` signals are valid (instructions have been executed). For instance, the source register values and the immediate value are only valid after executing an `BGE` instruction, hence can be wired to properties for comparison. Ideally, `*_valid` signals should be set and reset at the same time during model checking if there is no fault. If they are not equal, there is no need to prove other strobe properties.

```
assert_insn: assert property (
 golden_valid && faulty_valid |->
 faulty_insn == golden_insn );
```

The above is another example, which compares the retired instructions from the two cores. The retired instructions are stored in register `golden_insn` and register `faulty_insn`. These two registers are valid when `golden_valid` and `faulty_valid` are set. If `golden_valid` and `faulty_valid` are not set at the same time, the antecedent is not met, and there is no need to check the consequent. If the assertion fails, the injected fault has changed the executed instruction, causing a system failure. For instance, a fault is injected into the instruction FIFO altering the immediate value `imm_b_type` of the `BGE` instruction. The altered instruction is executed normally but produces a wrong branch result. Such a fault cannot fail architectural properties but can fail the strobe properties. The strobe properties should never fail if there is no fault, because the two cores should behave the same when there is no fault.

3.3.1.3 Summary and Comparison

After developing the architectural properties and strobe properties, we validate them using various approaches, as in Section 3.6.2. When proving either architectural or strobe properties, we assume an arbitrary fault exists in the (faulty) Ibex Core. The arbitrary fault is articulated by the implicit fault injection described in Section 3.2.3.

We tried both architectural properties and strobe properties to explore SEUs that may cause SDCs. After comparison, strobe properties can produce a higher fault coverage than architectural properties, mainly because architectural properties cannot cover faults in some structures such as Register File. However, the fault coverage is too high: some faults that cannot essentially cause SDCs are also identified as crucial faults by strobe properties. The detailed results and comparisons are shown in Section 3.8.

Apart from the above, the same structures (bits/registers) vulnerable to SEUs can be identified by the both groups of properties. However, there are some differences in terms of development effort and performance. It is simpler to develop the strobe properties because they just compare outputs. However, the Ibex Core is modified and duplicated, increasing the complexity of designing and verification. On the contrary, it is more difficult to develop (and validate) the architectural properties. However, we argued that it is a one-time cost because the architectural properties can be used for other RISC-V designs. The architectural properties can also provide a deeper insight into SDCs compared to the strobe properties.

3.3.2 Crash

Both the ISA manual and the Ibex Core design specification are referred to develop crash properties, which is different from developing the SDC properties. Some terminology must be explained first [6]:

- Exception: An unusual condition occurring at run time associated with an instruction in the current RISC-V hardware thread (hart). An exception can cause a crash.
- Interrupt: An external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control. An interrupt can cause a crash.
- Trap: The transfer of control to a trap handler caused by either an exception or an interrupt.

Both exceptions and interrupts can cause traps. By default, the Ibex Core operates in the machine mode. There are three modes in RISC-V: machine mode (M), supervisor mode (S), and user mode (U). Machine mode is the highest privilege mode in a RISC-V system. M-mode is used for low-level access to a hardware platform and is the first mode entered at reset [137]. We only explore SEUs when the Ibex Core is in the M mode.

All the interrupts in the Ibex Core are disabled, only exceptions in the M mode can cause traps/crashes. Rising and handling a trap relies on Control Status Registers (CSRs). This involves only the necessary hardware to develop crash properties by directly monitoring CSRs, compared to monitoring output ports. The RISC-V Instruction Set Manual specifies the mapping and function of CSRs [137]. RISC-V supports custom CSRs. There are custom CSRs in the Ibex Core. For example, a custom CSR named `cpuctrlsts` controls the runtime configuration of CPU components. Since the custom CSRs can vary in different designs, we do not consider them.

Some CSRs might be useful to debug traps. For example, a CSR named `mtval` stores exception-related information when an exception is encountered. This register can be used to assist software in handling exceptions. However, we did not choose this CSR because it can only store limited exception information. For example, there is an error in the load-store unit; a transaction is misaligned; and there is an illegal instruction exception. For all other exceptions, `mtval` stores no information and is set to 0.

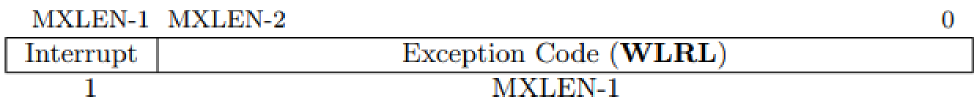


FIGURE 3.6: Machine Cause register `mcause` [6]

We found a CSR that meets our requirements. The CSR is named `mcause`. This register is an MXLEN-bit read-write register. MXLEN is 32 in the Ibex Core. `mcause` is formatted as shown in Figure 3.6. `mcause` stores the machine trap cause: the event code is stored in `mcause` when a crash is encountered. The Interrupt bit in Figure 3.6 is set if the trap is caused by an interrupt. However, all interrupts are disabled in the Ibex Core, so this bit is always 0. The Exception Code represents the exception (or interrupt if the Interrupt bit is set) that causes the crash. Only supported exception codes with corresponding crashes are listed in Table 3.3. 'Instruction access fault', 'Load access fault', and 'Store access fault' are terms for software exceptions; they are not hardware faults.

When a processor attempts to fetch an instruction from an inaccessible address, or from an address that does not exist, an instruction access fault exception is

TABLE 3.3: Exception Code in the Ibex Core

Exception Code	Description
1	Instruction access fault
2	Illegal instruction
3	Breakpoint
5	Load access fault
7	Store access fault
11	Environment call from M-Mode (ECALL)

encountered. When a processor attempts to execute an illegal or not-supported instruction, such as executing an atomic instruction in the Ibex Core, an illegal instruction exception is encountered. A breakpoint occurs if a processor meets an EBREAK instruction during execution. Similarly, an environment call from M mode occurs if a request to supervisor mode or user mode is made by a processor (running in M mode) executing an ECALL instruction. The EBREAK and ECALL instructions are two system instructions, which are used to access system functionality that might require privileged access [6]. The ECALL instruction is used to make a service request to the execution environment. The EBREAK instruction is used to return control to a debugging environment. A load access fault occurs if a processor attempts to read data from an inaccessible address or a nonexistent address in memory. Similarly, a store access fault occurs if a processor attempts to write data to an inaccessible address or a nonexistent address in memory.

Based on the function of the CSR `mcause`, crash properties can be developed. For example, to develop a property that can be used to identify faults causing store access faults, the antecedent could be 'the current privilege mode is the machine mode', and the consequent could be 'a store access fault should never occur (the exception code should never be 7)'. The following is an assertion specifying a crash caused by a store access fault. `crash_priv_mode` is the current privilege mode; `crash_priv_mode==2'b11` means the Ibex Core is operating in the machine mode. `crash_mcause_q` reads the exception code stored in the control status register `mcause`. `crash_mcause_q!=6'd7` specifies that the code of store access fault should never occur, hence the corresponding crash should never occur. The assertion `assert_store_access_fault` should never fail for the golden core. If the assertion fails, a fault causing a store access fault crash is found. The details of the identified crucial fault, including the fault location and the injection time, can be found in the auto-generated CEX.

```
assert_store_access_fault:  assert property (
 (crash_priv_mode==2'b11) |->
 (crash_mcause_q!=6'd7) );
```

3.3.3 Hang

To develop properties that can identify crucial faults causing hangs, scenarios for a hang must be classified first. There are three possible scenarios for a hang:

- WFI: The retired instruction is a WFI (Wait-For-Interrupt) instruction, which means the core will stay in the sleep state until it is activated again by other instructions. The waiting time is unsure, hence causing a hang.
- Dead State: The core can be thought of as a Finite State Machine (FSM). The FSM is stuck in a state and cannot leave that state. For instance, the FSM is frozen, hence causing a hang.
- Live State: The Finite State Machine (FSM) of the core is trapped in a state sequence. The FSM cannot return to a state from other states. For instance, the FSM is stuck in an infinite loop, hence causing a hang.

It is easy to model the first scenario as a property: the retired instruction should never be a WFI instruction. The following example is the hang (WFI) property. `valid` is true when the retired instruction is a valid RV32IMC instruction. `halt` is true if the retired instruction is the last instruction in the software. `!halt` is added to the antecedent to make sure the program is still running. `32'h10500073` is the WFI instruction.

```
assert_hang_WFI: assert property (
valid && !halt |->
(insn != 32'h10500073) );
```

It is not suitable to use the previous approach that identifies crucial faults by disproving correct behaviours to explore Dead States. It is easy to develop a property that specifies the next FSM state is not equal to the current FSM state. However, disproving the property assertion may produce false negatives. There exists a scenario where the FSM is designed to repeat a state several times. If a safe fault that causes no effect is injected under such a scenario, the assertion may still fail, hence a false negative.

An alternative approach is to use 'cover' instead of 'assert': specifying bad behaviours such as a Dead State as a property and using a 'cover' statement to find a scenario that matches the property, for example, create a sequence containing `$stable(an_FSM_state)`. `an_FSM_state` is an arbitrary FSM state used for demonstration. `$stable(an_FSM_state)` returns true if `an_FSM_state` has the same value at this clock cycle as it did in the previous clock cycle. Then develop a property that repeats the sequence infinite times using the repetition operator `[+]` (which repeats from 1 to infinite times).

The same problem exists in a Live State and the same solution can be applied to Live States, except additional AUX code may be required to monitor and record the repeated state sequence. In addition, the correct behaviour of ”the FSM can (eventually) return to a state (S0) from other states (S1,S2,S3...)” can be easily modelled as properties. However, this may create an infinite trace length.

It is possible to develop formal properties of the Dead State and Live State as shown in the above two paragraphs. In addition, JasperGold Superlint App can automatically generate corresponding properties of the Dead State and Live State. However, safety properties and liveness properties are involved. A safety property describes nothing bad happening. For instance, no deadlock. It is hard to formally verify the generated safety properties, because too many states are involved. A liveness property describes something good that eventually happens. For example, the FSM can eventually return to a state from other states (the FSM can eventually break the infinite loop). Liveness should be used with caution in formal verification because it is hard to fully verify liveness properties even with extreme time and high-performance hardware. In this research, we could not fully verify the liveness properties of our equipment. We only got undetermined results from the proof of liveness properties. For example, the liveness properties cannot be either proved or disproved after 24 hours. As a result, only the first scenario (WFI) is modelled as a formal property.

3.4 Step 3: Complexity Control Strategies

Besides the formal technologies implemented in the formal tool (whose details are in Section 3.5.2), we minimize state explosion and improve proof performance with the following complexity control strategies.

1. Black-boxing memories to reduce complexity.
2. Developing constraints to reduce state space and to avoid false negatives.
3. Verifying at a high level instead of in detail for each module.
4. Handling inclusive/undetermined results with case splitting and different proof algorithms/engines.

3.4.1 Black-Box

Firstly, the Instruction and Data Memories (IMEM and DMEM, Figure 1.1) are black-boxed. As shown in Figure 1.1, there are two memories that store instructions (IMEM) and data (DMEM) in the Ibex Core. Including memories in formal verification significantly increases design and verification complexity, which may lead to a state space explosion. It may also result in failing to fully prove assertions. It is necessary to abstract away irrelevant details to reduce the state space [138]. Excluding memories greatly reduces the design complexity and state explosion.

IMEM and DMEM are abstracted (black-boxed) by totally removing the corresponding source code. We are interested in bits inside the Ibex Core, not bits in memories. There have been many memory protection technologies. We assume all faults in IMEM and DMEM cannot propagate to the Ibex Core. Auxiliary (AUX) code in SystemVerilog is developed (according to the Ibex guidance) to replace IMEM and DMEM. The AUX code of the memories is simple: it handles only communication signals; it does not model any data or instructions.

For example, Figure 3.7 displays the memory transaction of IMEM. After receiving the request signal `req` and the instruction address `addr 1` and `addr 2` from the core, IMEM asserts the grant signal `gnt` immediately. Then after one clock cycle, IMEM outputs the corresponding instructions `instr 1` and `instr 2` and asserts the valid signal `valid` to the core.

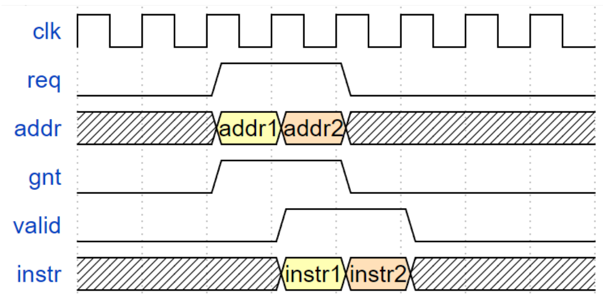


FIGURE 3.7: Memory Transaction of IMEM

3.4.2 Input Constraints

Secondly, constraints are developed as assumptions to speed up model checking and avoid false negatives. It is assumed that one clock cycle after receiving an instruction/data request, the abstracted memories output an instruction/data with a grant signal. Data from DMEM are unconstrained so that the model checker can explore all possible input data. However, there are some constraints on instructions from the IMEM. Only bit patterns corresponding to valid RV32IMC instructions are allowed. Another aim of these constraints is to make sure the Ibex Core works as expected when there are no faults. These constraints are expressed as SVA assumptions, as shown in the following.

```
assume_valid_RV32IMC: assume property (
(valid_RV32I || valid_RV32M || valid_RVC));
```

The above assumption limits the instruction to the core to be a valid RV32I instruction (valid_RV32I), a valid RV32M instruction (valid_RV32M) or a valid RVC instruction (valid_RVC). There are multiple valid bit patterns for an instruction. For instance, RV32I_LUI specifies the correct bit pattern of a Load Upper Immediate (LUI) instruction:

```
assign is_rv32_insn = (instr_rdata_i[1:0] == 2'b11);
assign RV32I_LUI = is_rv32_insn && (riscv_instr_opcode == 7'h37);
```

The LUI instruction places the immediate value (extracted from the instruction) to the highest 20 bits of the destination register and fills in the lowest 12 bits with zeros. instr_rdata_i is the instruction from IMEM to the core. is_rv32_insn is true if instr_rdata_i is a valid 32-bit RISC-V instruction. riscv_instr_opcode is the OPCODE of the instruction which determines the functionality of the instruction. 7'h37 refers to LUI.

These bit patterns are ORed together according to instruction sets. For instance, valid_RV32I is a 1-bit wire that ORs all RV32I bit patterns, as shown in the following. To save space, most bit patterns are not listed. The whole set of bit patterns is in Appendix F.

```
assign valid_RV32I = RV32I_LUI||RV32I_AUIPC||...;
```

The above statements make sure only valid RV32IMC instruction bit patterns are allowed during model checking. However, during our experiments, we observed that these constraints may lead to false negatives: errors and system failures can be encountered even without any crucial SEUs or with only safe SEUs. This is caused by valid instructions with illegal sequences (misaligned instructions). For example, the jump address of branch instructions must be aligned to a multiple of four, otherwise, the processor will try to align the instruction stored in the jump address by swapping the upper 16 bits with the lower 16 bits. Such instructions may be illegal, causing an exception. Therefore, instructions are constrained to be aligned with valid RV32IMC instructions, as shown in the following.

```
assume_address_by_4: assume property (
@(posedge clk_i) disable iff (!rst_ni)
aligned_by_4 && instr_gnt_i |->
(is_specified_RV32I || is_specified_RV32M) );

assume_address_by_2: assume property (
@(posedge clk_i) disable iff (!rst_ni)
!aligned_by_4 && instr_gnt_i |->
(is_specified_RV32C) );
```

These assumptions state that if the instruction address from the Ibex Core is aligned by 4, then it is an RV32IM instruction; otherwise, it is an RV32C instruction. This kind of problem could be caused by bad manually developed assembly language code. On the other hand, this kind of problem should never arise with a good compiler. With a qualified compiler, it is not necessary to worry about this problem, hence less effort is needed to develop instruction constraints.

Given the back-to-back memory transaction in the last subsection and the two pipeline stages in the Ibex Core, it can be figured out that most instructions take two clock cycles through the core. However, some multi-cycle instructions, such as branch, jump, load, store and multiplication and division, can stall the pipeline, and require more than two clock cycles to propagate through the core even with immediate memory response. On the other hand, we do not add more constraints than the above assumptions. For example, during model checking, the input instruction sequence may or may not contain a specific instruction, hence the latency of every instruction in the sequence is unknown. It is unnecessary to explicitly define the latency of every instruction as the time window for fault injection. We let the model checker explore all valid state space and decide the instructions and faults that lead to failures. If there exists such a crucial fault, the model checker finds the shortest CEX path where a fault is injected one to four clock cycles after reset, and an assertion failure arises after another at least two clock cycles. As a result, we do not develop extra assumptions of the fault injection time.

3.4.3 Verify at Architectural Level

Thirdly, we focus on architectural behaviour instead of detailed micro-architectural behaviour, such as state transitions. The architectural behaviour specifies the expected results of each instruction. Only inputs and outputs are required to verify the functionality. Moreover, the architectural behaviour is similar to that of other RISC-V designs because they all follow the same RISC-V ISA manual. Verifying architectural behaviour can treat the whole system as a black box, which allows abstraction in model checking.

3.4.4 Handle Undetermined Results

Finally, we use two approaches to handle undetermined results caused by state explosion. On the one hand, we use different engine modes to solve different problems, the details are in Section 3.5.2. On the other hand, case splitting is used to reduce the state space. If unbounded results could not be reported with the present fault space from a single model checking run, then we divided the fault space into smaller fault lists, and performed each model checking run to each fault list, as shown

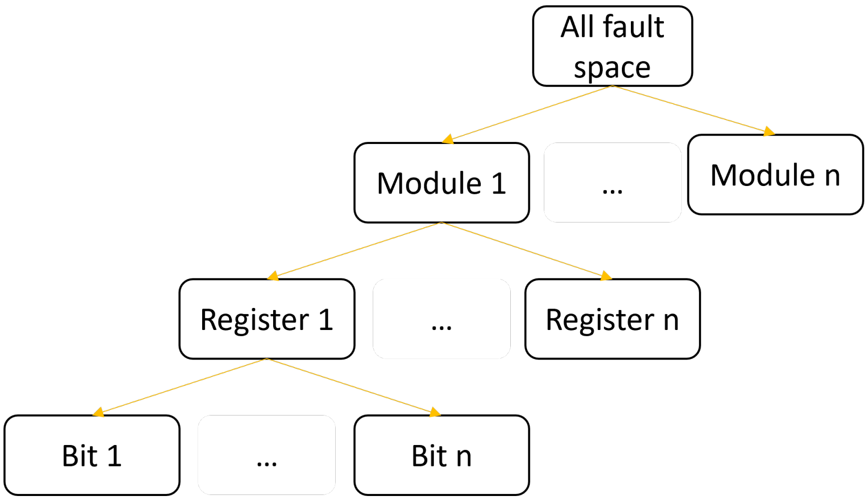


FIGURE 3.8: Case Splitting

in Figure 3.8 where a yellow arrow represents undetermined results and a rectangle represents a split fault list. This is done by SV assumptions. For example, if undetermined, we first divide the total fault space into module levels, where each fault list contains all faults in a single module. If such a fault list remains undetermined, we further divide it into register levels, where each fault list contains all faults in a single register. Most problems can be fully solved at the register level. The following is an example of splitting fault space into the register level. It is unnecessary to state `FI_index` is less than 2008 because of the register range.

```
assume_FI_index_stable_reg_1: //reg_1 contains bits [500:531]
assume property ( @(posedge clk_i)
$stable(FI_index) && (FI_index inside{[500:531]}) );
```

Case splitting is especially useful to solve undetermined results in step b) and step d) in Section 3.5.1, such as instead of attempting to prove all the remained faults are safe in one model checking run, dividing the remained faults into several sub groups. It is easier and faster to disprove or fully prove the fault list with a smaller size.

3.5 Step 4: Model Checking

Faults are identified by proving assertions and finding CEXs. We model the correct behaviour (such as a crash should never occur) as formal properties. We perform model checking on these assertions. If an assertion fails, a CEX will be given in the form of a waveform, such as a Value Change Dump (vcd) file that stores all digital logic traces. The fault attributes, such as fault location and injection time, can be found in the vcd file. Hence the faults can be extracted from the CEX. In this section, we firstly introduce our experimental strategy to improve proof efficiency by compressing repeated model checking runs and iteratively reducing the fault space. Then we provide settings of the formal tool including engine modes.

3.5.1 Experimental Strategy

In each model checking run, though there are multiple assertions, at most one CEX can be reported from each assertion. In other words, after a single model checking run with n assertions, at most n crucial faults can be identified; each assertion can only identify at most one crucial fault. The formal tool cannot report multiple CEXs from the same assertion at once, hence cannot identify multiple crucial SEUs (in different bits) violating the same assertion at once. As a result, for a given assertion, multiple model checking runs are required to identify all crucial SEUs in different bit locations, hence identifying all bits vulnerable to the error.

The most straightforward strategy is iterated model checking runs. In this case, 2008 runs are required to identify all SEUs in the Ibex Core, because there are 2008 bits in the Ibex Core. Each run explores SEUs from only one bit location. If a counterexample is reported, this bit is vulnerable to the error represented by the corresponding assertion. However, this strategy is still time-consuming. In addition, we observed that many model checking experiments produce the same results: target bits are not vulnerable to SEUs (SEUs in these bits have no effect). Performing iterated experiments on these bits is essentially a repetition of the same work, which wastes a lot of computation efforts and time.

Therefore, the following experimental strategy is developed to explore SEUs:

> For each assertion:
>
> a) Start with an arbitrary SEU;
>
> b) Perform model checking to identify a crucial SEU that violates the assertion;
>
> c) Record the crucial SEU and remove it from the fault space;
>
> d) Go back to b) with the updated fault space. Repeat until no more crucial SEU can be found and the assertion is fully proven.

The strategy starts with an arbitrary SEU: both the fault location and the injection time are stable free nets so that the fault space contains all possible SEUs. Details of using implicit fault injection to inject an arbitrary SEU are in Section 3.2.3. In addition, there is no other constraint, such as a specific time window for fault injection, as explained in Section 3.4.

The second step is to perform one model checking run to identify a crucial SEU that violates the given assertion. Details of the properties are described in Section 3.3. Details of the formal tool configurations are in the next subsection.

COI analysis is performed in parallel with model checking. COI has been explained in Section 2.4. The COI includes all bits that may affect the target property. Outside the COI are all bits that have no effects on the target property. Faults that occur in the bits outside the COI will never fail the target property assertion, hence these bits are structurally safe bits. By performing COI analysis, 81 common structurally safe bits for all COI were found. It is unnecessary to explore faults in these structurally safe bits using model checking, which saves time and effort.

The third step is to add the identified crucial SEU from the last step to constraints (as SVA assumptions) in order to exclude it from further model checking runs. For example, after finding one crucial SEU violating the assertion, the vulnerable bit is excluded from the fault space the next time proving the same assertion. We assume after identifying a crucial SEU, the vulnerable bit will be protected. This step aims to avoid repeatedly identifying the same SEU. It also reduces the fault space and forces the formal tool to explore the other SEUs. Each assertion has a unique fault list, because different assertions can identify different crucial faults. The following is an assumption example about a crash assertion after identifying two crucial SEUs.

```
`ifdef insn_access
  assume_FI_index_stable:
  assume property (
  $stable(FI_index) && (FI_index<2008)
  && !(FI_index inside {608,609}) );
`endif
```

We use the range from 0 to 2007 to represent all the register bits in the Ibex Core. The bit index is shown in Appendix E. In the above example, the already identified crucial SEUs are in bits 608 and 609. These two bits are in the register `err_q`, which determines the validity in the `FIFO` module. `FI_index` is the fault injection bit location. `!(FI_index inside {608,609}))` makes sure the already identified crucial SEUs can no longer be explored in further model checking runs.

The final step is to go back to step b) to run a different model checking with the updated assumptions. We repeat until the assertion is fully proven, which means the remaining SEUs in the fault space have no effect. Then move to other assertions. At this point, all crucial SEUs have been identified (and added to the assumptions). Ideally, only one final model checking run is performed to confirm that all the remaining SEUs have no effect. However, for most assertions the final model checking run cannot report unbounded results, hence case splitting as described in subsection

3.4.4 is used. Case splitting splits the fault space and adds some extra model checking runs in order to get unbounded results. Nevertheless, compared to exploring each SEU one by one, identifying all safe SEUs at once or with case splitting significantly saves a lot of time and effort. Efforts and results of case splitting will be given in Section 3.8.

3.5.2 Configurations of Cadence JasperGold

There are multiple formal techniques to improve proof performance and to handle state explosion, as mentioned in Section 2.4. Some of the techniques have been integrated into Cadence JasperGold. For example, JasperGold includes both SAT- and BDD-based proof engines with variations of these algorithms. In Section 1.4, the formal tool and the proof engines used in this research have been briefly introduced. We limited the proof time to three hours for each model checking run. We did not limit the proof trace. We enabled ProofMaster to improve proof performance. We ran two FPVs in parallel in one server, hence we set the maximum number of jobs to 10. We assigned one thread to each engine because each core in the server only has one thread. We chose specific engine modes to solve different types of problems. In the following, we explain which engine modes are chosen to solve which problems, and why these engines are chosen.

Single-property engines were chosen because there is only one assertion in each model checking run, as shown in subsection 3.5.1. Though engines Hts and Hps are basic engines, they are the most used engines in this research. Step b) in subsection 3.5.1 focuses on finding counterexamples. Hence we need to choose engines that are good at finding traces. After experiments, we found engine Hts was able to report 99% of counterexamples within a short time (from seconds to minutes). The remaining 1% counterexamples cannot be found by Hts within the time limit. But they can be found with other Engines such as B and AD (some counterexamples needed case splitting to reduce the fault space). In this research, within the same time, Hts is faster than the other engines that focus on finding counterexamples in most cases; only B can outperform Hts in some cases. As a result, we chose engines Hts, B and AD to find counterexamples.

Step d) in subsection 3.5.1 focuses on full proof. Hence engines that aim at proof were chosen. Engine Hps was able to fully prove the most simple properties/problems, such as those architectural properties that model base integer computational instructions, crash properties and hang properties. Engine Hps was not the only engine that could solve these problems, but it was much faster (from seconds to minutes) than other engines to solve the same simple problems.

It was much harder for Hps to prove branch & jump properties, and the most difficult to prove load & store properties. Hps cannot fully prove them even after 25 hours. We tried other engines to handle these complex problems. For instance, using Engines N and Tri to fully prove single-properties and Engine R to fully prove multi-properties. They can solve part of the problems within hours. We also tried different engine modes, because some engine combinations can speed up the proof process. For example, Engines I, C and C2 use abstractions to iteratively include logic from the COI and should be suitable for problems when the analysis region becomes too complex. However, we ran out of memory and this engine mode could not fully prove or find a counterexample after 24 hours. Finally, we tried Abstraction Engines AD and AM. AD was able to find counterexamples that are hard to reach. AM with case splitting can solve the remained problems: most problems took minutes but a few hard-to-solve cases took up to 10 hours. As a result, we chose Hps to fully prove the most properties, and N, R and (mainly) AM for the other hard-to-prove properties.

3.6 Validating Fault Injection and Properties

3.6.1 Validate Fault Injection

Simulation is used to validate the fault injection mechanism. CoreMark, a benchmark that is designed to test the functionality of a processor core, is chosen as the testbench. Verilator is used to perform simulation. Verilator is an open-source cycle-accurate simulator that supports SystemVerilog. Verilator converts the design into a C++ or SystemC model. A wrapper file written in C++ that includes the $main()$ function is used to instantiate and evaluate the model. Verilator uses both the converted model and the wrapper file for simulation. The Ibex Core project provides the wrapper file for simulation. The wrapper file is modified, by adding two command arguments (the fault location and the fault injection time). When there is no fault, the benchmark passes without errors, which proves the fault injection mechanism has no functional impact on the Ibex Core when there is no fault. Then we perform a random fault injection with 1000 faults in the Ibex Core.

TABLE 3.4: Results of Random Fault Injection

Fault Effects	# of Faults	Time (s)
No effect	947	9
SDC	19	9 to 10
Crash	23	<9
Hang	11	>100

Table 3.4 lists the results of the random fault injection results. Among the 1000 injected faults, 947 faults have no effect, 23 faults lead to crashes (16 are caused by illegal

instructions), 19 faults result in SDC and 11 faults cause hangs. Most of the simulations (correct results and SDC) of the random fault injection took about 9 to 10 seconds. Simulations that crash took less than 9 seconds. Simulations that are longer than 4194660 clock cycles (which is ten times the golden run time) were regarded as timeout (hangs). In total, it took around three days to test the 1000 faults. The results prove the fault injection mechanism can successfully inject faults into the core and cause errors.

3.6.2 Validate Properties

The developed properties are validated using both simulation and model checking. Firstly, a fault-free simulation with CoreMark is performed to collate assertion-based coverage data. Assertion-based coverage measures how many assertions got activated [139]. The coverage data shows all the assertions are activated at least once, which means no vacuous passes. A vacuous pass happens if the antecedent of an assertion fails, then the assertion will be proven vacuously without checking the consequent. A vacuous pass means the assertion fails to validate the design. In parallel, we perform model checking on the Ibex Core with no faults. The model checking result also shows no vacuous passes. In addition, both simulation and model checking results contain no assertion failures: all the assertions are proven, which means the properties violate no design specifications. No vacuous passes and no assertion failures prove the properties successfully follow the design specifications.

Then we perform model checking again with manually injected faults (mutations). The faults are injected by assigning certain values to the two fault control signals through assumptions. For example, the two assumptions about `FI_time` and `FI_index` are modified as follows. `FI_time` can be assigned to a clock cycle number. `FI_index` can only be a number between 0 and 2007 because there are 2008 bits in the Ibex Core. The following assumptions produce a concrete fault/mutation. The following example is only used for property validation; the following two assumptions must not be used in the actual model checking experiments to identify and classify SEUs.

```
assume_FI_time_stable:
assume property ( @(posedge clk_i) FI_time == 10 );
//10 can be changed to any number

assume_FI_index_stable:
assume property ( @(posedge clk_i) FI_index == 1 );
//1 can be between 0 and 2007
```

All the manually injected faults (mutations) can be reported, because all the assertions can fail if a crucial fault is injected inside the corresponding COI bits. For example, if a fault is injected into the Program Counter, then an assertion monitoring a crash fails. By analysing the counterexample, the input fault control signals can be found, hence the injected fault can be identified. The results prove the properties can identify mutations and, hence can be used to explore SEUs.

3.7 Validating Framework

After validating the fault injection mechanism and the developed properties, we validate the whole framework by applying the proposed method to two well-known fault-tolerant technologies: Triple Modular Redundancy (TMR) and Shadow Registers (SR). In the following, we first introduce the two fault-tolerant technologies and how they are implemented in the Ibex Core, and then we introduce how to use them to validate the framework. The same method, experiment setup and tool setting mentioned above are used for framework validation.

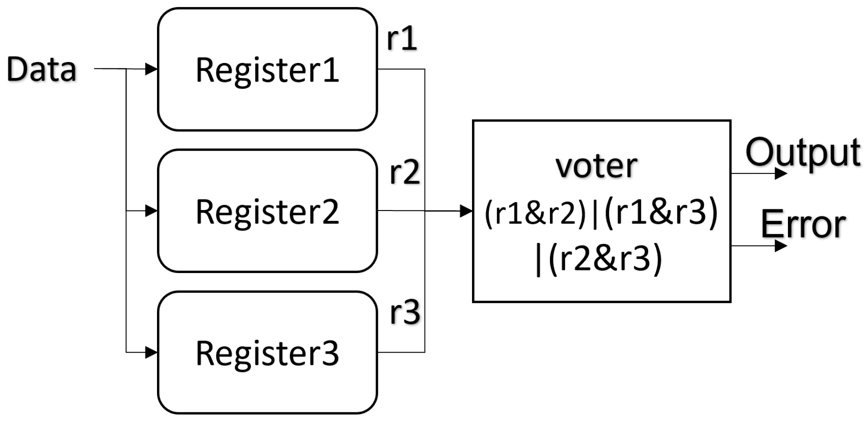


FIGURE 3.9: TMR in the Ibex Core

TMR triplicates the design under protection and adds a voter after the triplication. TMR can detect double faults and correct single faults. The design under protection can be a system, a module, a register, or a bit. In this work, as shown in Figure 3.9, TMR is introduced at the register level: each register in the Ibex Core is triplicated. The protected registers include the Register File but exclude the memories, Figure 1.1. We assume the memories are fault-protected hence all the faults in the memories cannot propagate to the core part.

Extra hardware, such as double registers, are added to the design by implementing TMR. It is possible that SEUs occur in the extra registers instead of the original register. To cover all possibilities, we evaluated faults in both the original bits and the extra bits. We assume faults could occur in both the original bits and the extra bits equally.

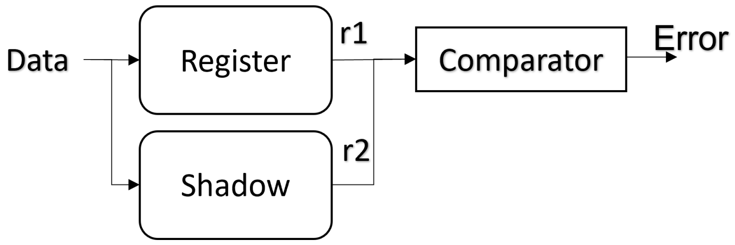


FIGURE 3.10: Block Diagram of Shadow Registers in Ibex

A majority voter is added after each triplicated register. The voter is used to detect and correct faults. If a single fault is injected into any of the registers in Figure 3.9, the voter can decide the correct output. The following equation shows the principle of the majority voter, $r0 = (r1\&r2)|(r1\&r3)|(r2\&r3)$, where $r0$ is the correct output decided by the voter, $r1$, $r2$, and $r3$ are three inputs from the original design and two copies. & means AND, | means OR. The majority voter is a simple AND–OR circuit: the voter outputs 1 if two or more inputs are 1; and 0 if two or more inputs are 0.

Similar to TMR, SR enhance the protected register by adding a shadow copy and placing a comparator after them, as shown in Figure 3.10. A fault could occur in either the original register or the shadow copy. Both cases are considered in this work.

SR can only detect single faults or double/multiple faults that occur in the same register. SR cannot correct faults by itself. Extra fault correction technologies are required to correct detected faults. We assume extra fault correction technologies will be implemented after the analysis, hence all the detected faults will be corrected.

3.7.1 Fault Detection

If a single fault is injected, causing a mismatch between the original register and its copies, an error signal will be set by the corresponding TMR voter. There are 71 registers in the Ibex Core, hence there are 71 voters and 71 error signals ($error\_reg1...error\_reg71$). The 71 error signals are ORed together to indicate a fault has been detected by the TMR in the Ibex Core:
$tmr\_error = error\_reg1||error\_reg2||...||error\_reg71$.
If any voter in the design detects a fault, `tmr_error` sets to logic high.

Similarly, SR raises an error indication signal if a mismatch caused by a fault is found by the comparator: $sr\_error = error\_reg1||error\_reg2||...||error\_reg71$. Though the same error signal names are used, these error signals are different from the error indication signals from TMR.

Since there is much in common between TMR and SR, such as both can detect and correct single faults (we assume extra fault correction technologies will be implemented with SR to correct faults), we only demonstrate TMR as the example in the following paragraphs. However, we did perform the following method and experiments on both TMR and SR to validate the framework.

Each error signal in TMR has a unique property that checks if the injected fault can be detected by the corresponding voter. Such properties are named detection properties. These detection properties are used to evaluate the fault detection effectiveness of TMR. The following is an example of TMR.

```
property p_TMR_reg1;
  !error_reg1;
endproperty
```

$error\_reg1$ is the error signal from the TMR enhanced register $reg1$ in the register file. If the property assertion fails, a fault injected in any of the triple registers can be detected by TMR. There are 71 registers in the Ibex Core, hence there are 71 detection properties. The number of detection property assertion failures is used to determine the detection effectiveness:

$detectioneffectiveness = \frac{detectionassertionfailures}{71}$

Ideally, all the SEUs should be detected by TMR; all the 71 assertions should fail with an arbitrary fault; hence 100% detection efficiency. Otherwise, there may be a bug in the framework, and further analysis is required to correct the framework. For example, the property `p_TMR_reg1` cannot fail even if a fault is injected to the register `reg1`. This may be a real design bug, such as the inputs of the TMR voter were wired together; or a framework bug, such as incorrect development or implementation of FI/assumptions/properties.

3.7.2 Fault Correction: SDC, Crash and Hang

Strobe properties are used to identify SEUs that can cause SDCs in the presence of TMR. Details of strobe properties have been given in Section 3.3.1.2. We chose strobe properties rather than architectural properties because we found that it is faster to use strobe properties for simple designs such as the Ibex Core. There are two differences compared to Section 3.3.1.2. The first difference is that both cores are protected by TMR. The second difference is that the error indication signal `tmr_error` is added into antecedent in each property assertion. If the error indication signal is not set, which means no error is detected by TMR, then it is unnecessary to check the consequent of the property. The following is an example.

```
assert_rvfi_insn: assert property (
 golden_rvfi_valid && faulty_rvfi_valid && tmr_error|->
 faulty_rvfi_insn == golden_rvfi_insn );
```

The example checks whether the detected fault can alter the instruction, hence causing an SDC, when the design is under the protection of TMR. A proven assertion (without a vacuous pass) means all the detected faults by TMR cannot alter the retired instruction just as expected, which further proves the correctness of the framework.

Similarly, the error indication signal `tmr_error` is also added into antecedent in crash properties in Section 3.3.2 and hang properties in Section 3.3.3.

The following is an example of a modified crash property assertion. This assertion specifies that if a fault has been detected by TMR, then the detected fault should never cause a store access fault crash.

```
assert_store_access_fault:  assert property (
 (crash_priv_mode==2'b11) && tmr_error |->
 (crash_mcause_q!=6'd7) );
```

The following is the modified hang property assertion. The assertion specifies that if a fault is detected by TMR, the retired instruction is a valid RV32IMC instruction, and the retired instruction is not the last one in the software program (the program has not ended yet), then the core should not be stuck in a sleeping state that waits for a new instruction.

```
assert_hang: assert property (
valid && !halt && tmr_error |->
(insn != 32'h10500073) );
```

If any of the above three assertions fails, a fault that can be detected but cannot be corrected by TMR is found, which is in fact impossible, as TMR can correct all the detected faults. The false negatives could be caused by either a design bug, such as incorrect wiring; or a framework bug, such as incorrect FI, assumptions and properties. If all assertions are proven, there is no bug in the design or the framework. As a result, the framework can be validated by evaluating TMR and SR.

3.7.3 Validation Results

TMR adds 4016 bits and 71 combinational voters to the Ibex Core. Without any faults, the enhanced Ibex Core works as expected. All the detection assertions are proven, and all the assertions without the error indication signal are proven. We did not use the modified properties in the last subsection, because the error indication signals were not set without faults, which can lead to a vacuous pass when proving assertions with the error indication signals in the last subsection. This result indicates that the implemented TMR does not influence the functionality of the Ibex Core when there is no fault.

With an arbitrary SEU, all the error indication assertions fail. All the TMR voters can raise an error indication signal immediately, if a fault is injected into the corresponding enhanced bits. All the injected faults can be detected, the fault detection efficiency is hence 100%. All the strobe, crash and hang assertions are proven without a vacuous pass. The fault correction efficiency is 100%. All the above results (both the fault detection efficiency and the fault correction efficiency) satisfy the features of TMR, which validate the framework.

Similarly, SR adds 2008 bits and 71 comparators to the Ibex Core. Table 3.5 shows the results of evaluating SR using the framework. The second column, 'Detected' lists the number of detected bit locations by SR. The results show that faults in all bit locations, including both the original bits and the extra shadow copies, can be detected by SR. Hence the fault detection effectiveness of SR is 100%.

TABLE 3.5: Evaluating Results of Shadow Registers

Group	Detected	Failure	Proven
SDC	4016	42540	17700
Crash	4016	318	23778
Hang	4016	20	3996

There are 45226 SDC failures in Table 3.5. SR can detect crucial SEUs that may cause SDCs in 4016 bit locations. The 4016 bit locations include both the original bits and the shadow copies. The Ibex Core bits that are vulnerable to SDCs without fault-tolerant technologies will be given in subsection 3.8.1.1, where 21270 crucial faults causing SDCs are identified by strobe properties. The 21270 crucial faults are also included in Table 3.5. SR doubles the bits, hence the vulnerable bits are also doubled. That is why the number of SR-detected crucial SEUs to SDCs, crashes and hangs in Table 3.5 are doubled compared to those in subsection 3.8.1.2 and 3.8.1.3. Hence the crucial fault detection efficiency is 100%. The results match the features of SR: SR can detect all SEUs including all crucial SEUs in the Ibex Core, hence the results can validate the framework.

In summary, we use our method to evaluate TMR and SR. Though the results are unsurprising, the validation results validate the whole framework. We further expand the method to evaluate a different fault-tolerant technology in Chapter 4.

3.8 Results and Analysis

All the model checking results were either 'Proven' or 'Failure' (except the RV32M properties). On average, it only took one and a half minutes to report each of the 99% counterexamples (step b in Section 3.5.1). The remaining 1% were disproved with case splitting. Each case took one minute to be disproved. It took three hours to fully prove each of the crash and hang properties. It took one hour to fully prove each of the RV32IC Integer Computational Instruction properties without case splitting. Case splitting was used to fully the RV32IC Control Transfer and Load and Store Instruction properties. 80% of the split cases were fully proved by one and a half hours. However, it took three hours to fully prove 19% of cases, and it took five hours to fully prove 1% of hard-to-prove cases.

In this section, we interpret the results at two levels: hardware-level and instruction-level. Based on the SEU effects, the corresponding bits can be classified into safe bits (SEUs in these bits have no effects), and vulnerable bits to SDCs/crashes/hangs (SEUs in these bits can cause SDCs/crashes/hangs). Similar for instructions. In each subsection, we first explain the safe part, then the vulnerable part. We introduce not only faults, but also the corresponding structures (bits, registers, modules) and instructions. Hardware-level identifies bits vulnerable to SEUs; hardware protection technologies, such as TMR, can be implemented to protect these vulnerable bits. Instruction-level identifies instructions vulnerable to SEUs; software protection technologies, such as redundant instructions, can be implemented to protect these vulnerable instructions.

3.8.1 Hardware-level

3.8.1.1 SDC

In this subsection, we first explain the results of proving architectural properties, then the results of proving strobe properties. Finally, we compare both results, i.e., in the term of fault coverage ratios.

Architectural Properties Table 3.6 lists the results of proving architectural properties. The columns called *Name* list property names. They also represent the monitored instructions. The columns called *Failure* and *Proven* list the number of vulnerable bits and safe bits to the corresponding instructions. For example, as shown in the first row, there are 91 bits that may lead to malfunctioning of the `RV32I_LUI` instruction, such as a wrong read value, hence causing an SDC. On the other hand, the other 1917 bits in the Ibex Core cannot corrupt the `RV32I_LUI` instruction and cause an SDC, hence safe bits.

Table 3.6 divides the architectural properties to different groups based on the instruction functions: RV32IC Integer Computational Instructions, RV32IC Control Transfer Instructions, RV32IC Load and Store Instructions and RV32M Instructions. There is no undetermined result for RV32IC instructions. All the RV32IC properties in Table 3.6 are fully proved. It took the least time and effort to fully prove the RV32IC

TABLE 3.6: Using Architectural Properties to identify faults causing SDCs

Name	Failure	Proven	Name	Failure	Proven
RV32I_LUI	91	1917	RVC_LUI	130	1878
RV32I_ADD	113	1895	RVC_ADD	126	1882
RV32I_ADDI	112	1896	RVC_ADDI	129	1879
RV32I_XOR	113	1895	RVC_XOR	128	1880
RV32I_XORI	113	1895			
RV32I_OR	112	1896	RVC_OR	128	1880
RV32I_ORI	112	1896			
RV32I_AND	112	1896	RVC_AND	126	1882
RV32I_ANDI	112	1896	RVC_ANDI	130	1878
RV32I_SLL	113	1895			
RV32I_SLLI	113	1895	RVC_SLLI	127	1881
RV32I_SRL	113	1895			
RV32I_SRLI	118	1890	RVC_SRLI	131	1877
RV32I_SUB	107	1901	RVC_SUB	129	1879
RV32I_SRA	106	1902			
RV32I_SRAI	115	1893	RVC_SRAI	134	1874
RV32I_AUIPC	91	1917	RVC_LI	130	1878
RV32I_SLT	112	1896	RVC_MV	132	1876
RV32I_SLTI	112	1896	RVC_ADDI4SPN	134	1874
RV32I_SLTIU	112	1896	RVC_ADDI16SP	129	1879
RV32I_SLTU	106	1902			
RV32I_JAL	121	1887	RVC_JAL	134	1874
RV32I_JALR	81	1927	RVC_JALR	90	1918
RV32I_BEQ	114	1894	RVC_BEQZ	130	1878
RV32I_BNE	107	1901	RVC_BNEZ	121	1887
RV32I_BLT	107	1901	RVC_JR	91	1917
RV32I_BGE	115	1893	RVC_J	128	1880
RV32I_BLTU	105	1903			
RV32I_BGEU	115	1893			
RV32I_LW	106	1902	RVC_LW	133	1875
RV32I_LB	109	1899	RVC_LWSP	132	1876
RV32I_LBU	108	1900			
RV32I_LH	107	1901			
RV32I_LHU	109	1899			
RV32I_SW	106	1902	RVC_SW	131	1877
RV32I_SB	106	1902	RVC_SWSP	120	1888
RV32I_SH	106	1902			
RV32M_MUL	1128	81	RV32M_MULHSU	1129	81
RV32M_MULH	1128	81	RV32M_MULHU	1128	81

Integer Computational Instruction properties. Each RV32IC Integer Computational Instruction property was fully proved by the engine Hps without case splitting within one hour. It took more time and effort to fully prove the RV32IC Control Transfer Instruction properties. The jump instructions were fully verified by Hps with case splitting. Each case took about one hour. The branch instructions were harder to fully prove by Hps even with case splitting, hence AM was added to the engine mode.

After splitting the fault space of the branch instructions, most cases were fully proved by AM within three hours, a few were fully proved by Hps. At this point, the fault space of most properties was only split into register levels. It was the most difficult to fully prove the RV32IC Load and Store Instructions. The fault space was split into bit levels, where each fault list contains only a few bits or even one bit. We set the time limit to 24 hours for the load and store cases. Most case was fully proved by AM within three hours. A few cases even took five hours to be fully proved.

However, due to the challenge of formally verifying multipliers and dividers, we cannot fully verify RV32M architectural properties. We identified some crucial SEUs to the multiplication instructions, but we could not fully prove any safe SEU to the multiplication instructions with the method. We could not even identify any crucial or safe SEU to division instructions, because the divider in the Ibex Core takes far more clock cycles than the multiplier. We also tried to abstract the multiplier and the divider using built-in functions of FSV but it did not work. Details will be discussed in Section 3.9. The only safe SEUs to RV32M properties were identified by FSV COI analysis.

We performed FSV COI analysis and identified 81 common structural safe bits which cannot result in SDCs. The safe bits are: `instr_rdata_alu_id_o[24:15]`, `instr_rdata_alu_id_o[11:7]`, `stored_addr_q[31:0]`, `fetch_addr_q[31:0]`, `imd_val_q[1][33:32]`. These bits are reliable because all of them are not consumed (i.e. further used) in the Ibex Core. In the Ibex Core, the instruction is duplicated and stored in two 32-bit registers to reduce fan-out. One is the register `instr_rdata_alu_id_o` in `if_stage` module. This register stores the instruction to the ALU in the Ibex Core. The ALU only reads specific bit fields in `instr_rdata_alu_id_o`, such as OPCODE. The other bit fields [24:15] & [11:7] are not consumed and hence safe.

Similarly, register `stored_addr_q` in `prefetch_buffer` module stores the request address in memory. Since the address is aligned by 4, the two least significant bits are not consumed, hence faults in the `stored_addr_q[1:0]` have no effect. It may be argued that apart from the two least significant bits in the registers `stored_addr_q` and `fetch_addr_q` in the `prefetch_buffer` module, the other bits can be vulnerable to SEUs. These bits decide the request instruction address to the memory. A fault in these bits may lead to a wrong instruction address, hence an invalid instruction may be produced by the memory. A system failure may occur due to an invalid instruction. However, such failure is out of consideration. This work focuses on bits inside the Ibex Core. We assume the memories cannot propagate any errors, all the errors in the memories will be corrected. As stated in Section 3.4, we abstracted the memories by totally removing the corresponding code, hence the memories cannot store any data (whether including faults or not) from the core part. We developed assumptions to make sure all the instructions from the instruction memory are valid and aligned instructions. In other words, though the faults in the registers `stored_addr_q` and `fetch_addr_q` can propagate to the memory, the faults are corrected by the memory and cannot propagate from the memory. Hence, all the bits in the registers `stored_addr_q` and `fetch_addr_q` cannot produce errors and are safe bits.

Based on the model checking results, common safe bits/registers/structures to architectural properties can be identified, as shown in Table 3.7. Since we could not solve all the undetermined results of RV32M properties, only RV32IC results are discussed (RV32M is excluded from the following results). There are three reasons why these structures are safe. 1) Some bits are not consumed with the given assumptions, such as the `MULTDIV` module is not triggered when there is no RV32M instruction and `imd_val_q` is only used in the `MULTDIV`. 2) SEUs in some structures, such as all registers (except register `ctrl_fsm_cs` which stores the current state of the module) in the `controller` module and all registers in the `CSR` module, can change the privilege mode so that the core is no longer in the M mode; we only construct failures in the M mode as properties, as stated in section 3.3. 3) Faults are hidden due to topology, which is a limitation of architectural properties, the details are in the next paragraph.

TABLE 3.7: Common Safe Structures to Architectural Properties

Bits	Registers	Module
[4:11]	All except ctrl_fsm_cs	controller
[12:477]	All	CSR
[512:607]	rdata_q	FIFO
[611:678]	imd_val_q	id_stage
683	instr_new_id_q	if_stage
[723:727]	instr_rdata_alu_id_o	if_stage
[731:740]	instr_rdata_alu_id_o	if_stage
749	instr_fetch_err_plus2_o	if_stage
[800:823]	rdata_q	load_store_unit
[824:825]	rdata_offset_q	load_store_unit
[830:861]	addr_last_q	load_store_unit
865	handle_misaligned_q	load_store_unit
[868:942]	All	MULTDIV
[943:1012]	All except rdata_pmp_err_q	prefetch_buffer
[1015:2006]	All	RegFile
2007	core_busy_q	Top

Instructions are important in the core. However, the register `rdata_q` that stores the instructions in the `FIFO` module is identified as a safe register in Table 3.7. Architectural properties read retired instructions from the core. If the instruction is fetched with a fault from the `FIFO`, in the Ibex Core there is no safety mechanism to detect or correct the fault, then architectural properties can only calculate theoretical values based on the faulty retired instruction. However, there is no way for architectural properties to generate correct theoretical values without a correct retired instruction. As a result, faults in the `rdata_q` in the `FIFO` module can never be identified due to topology. Similarly for the listed registers in the `load_store_unit` module and the `RegFile` module in Table 3.7. This limitation motivated us to develop strobe properties, as mentioned in subsection 3.3.1.1.

Table 3.8 lists the crucial SEUs and vulnerable hardware structures to architectural properties. In counterexamples, most of the identified crucial SEUs were injected at the fifth clock cycle after reset, when the FSM in the `controller` enters DECODE state (the normal operation state). The other identified SEUs were injected at later clock cycles such as the seventh clock cycle, because a longer input sequence was required to reach corner cases. Then after at least two clock cycles, an assertion failed, depending on whether the pipeline was stalled. Most of the counterexamples were reported by Hts without case splitting. Then with case splitting, 61 hard-to-reach crucial SEUs were identified by Hts and AD within minutes. These 61 SEUs are identified by proving properties about RV32IC Control Transfer Instructions and RV32IC Load and Store Instructions, because they are more complex than RC32IC Integer Computational Instructions.

There are multiple architectural properties. Some properties can report the same crucial SEUs, hence the corresponding bits/registers/modules are more vulnerable than others. We divide Table 3.8 into three blocks based on the frequency of crucial SEUs. In the first block labelled with $\geq$50, SEUs in the bits can violate more than 50 architectural properties. In the second block, SEUs in the bits can violate 20 to 50 architectural properties. SEUs in the last block labelled with $\leq$20 can violate no more than 20 architectural properties. Table 3.8 can be used to rank the reliability of bits/registers/modules in the Ibex Core. Fault-tolerant technologies should be implemented in vulnerable structures to improve hardware reliability.

The most crucial SEUs occur in the first block of Table 3.8. SEUs in PC-and instruction-related registers cause the most architectural property failures. Registers `instr_addr_q` and `pc_id_o` store the current PC. In fact, the value in `pc_id_o` is updated by the value in `instr_addr_q`. The `FIFO` outputs are fed into a `compressed_decoder`, which decompresses the instruction and stores it in the register `instr_rdata_id_o` in the `if_stage` module. The same decompressed instruction is also duplicated in the register `instr_rdata_alu_id_o` to reduce fan-out. The `instr_rdata_id_o` is then fed into both the `controller` and the `decoder`, the `instr_rdata_alu_id_o` is only fed into the `decoder`. That is why the vulnerable bit fields in the two registers are not the same. The `branch_set_raw_q` indicates a branch. A fault in this register can alter the `controller` and output a wrong PC, hence causing an assertion failure. The `instr_valid_id_q` stores the validness of the instruction to the second pipeline stage. The `instr_fetch_err_o` and `illegal_c_insn_id_o` are similar to the `instr_valid_id_q`: they store the validness of the fetched instruction and the validness of the compressed instruction to the second pipeline stage. A fault in these three registers can alter both the `controller` (the FSM of the core) and the

TABLE 3.8: Crucial SEUs with structures to Architectural Properties

SEUs	Bits	Registers	Module
≥50	[478:508]	instr_addr_q	FIFO
	679	branch_set_raw_q	id_stage
	682	instr_valid_id_q	if_stage
	[684:690]	instr_rdata_id_o	if_stage
	[710:715]	instr_rdata_id_o	if_stage
	[716:722]	instr_rdata_alu_id_o	if_stage
	748	instr_fetch_err_o	if_stage
	767	illegal_c_insn_id_o	if_stage
	[768:799]	pc_id_o	if_stage
	[862:864]	ls_fsm_cs	load_store_unit
<50	[0:3]	ctrl_fsm_cs	controller
>20	[509:511]	valid_q	FIFO
	[608:610]	err_q	FIFO
	681	id_fsm_q	id_stage
	[691:698]	instr_rdata_id_o	if_stage
	[704:709]	instr_rdata_id_o	if_stage
	[728:730]	instr_rdata_alu_id_o	if_stage
	[750:765]	instr_rdata_c_id_o	if_stage
	766	instr_is_compressed_id_o	if_stage
	866	pmp_err_q	load_store_unit
	[1013:1014]	rdata_pmp_err_q	prefetch_buffer
≤20	680	branch_jump_set_done_q	id_stage
	[699:703]	instr_rdata_id_o	if_stage
	[741:747]	instr_rdata_alu_id_o	if_stage
	[826:827]	data_type_q	load_store_unit
	828	data_sign_ext_q	load_store_unit
	829	data_we_q	load_store_unit
	867	lsu_err_q	load_store_unit

decoding process (such as failing to read correct source register address), hence causing an assertion failure. The `ls_fsm_cs` stores the FSM state in the `load_store_unit` module which takes care of accessing the data memory.

In the second block of Table 3.8, the `valid_q` determines the validness of all the three entries in the `FIFO`. The `error_q` stores the instruction fetch error (which is valid on the data phase of a request) from the IMEM. The `id_fsm_q` stores the current FSM state of the `id_stage` module. The `instr_rdata_c_id_o` stores the compressed instruction. The `instr_is_compressed_id_o` determines whether the instruction is 16 bits (compressed) or 32 bits. The Ibex Core supports an optional Physical Memory Protection (PMP) unit, which implements a region-based memory access checking protocol. However, PMP is disabled by default and in this work. The `pmp_err_q` indicates the PMP has detected an error. Ideally `pmp_err_q` should be always zero, because there is no PMP in the Ibex Core. Similar for the `rdata_pmp_err_q`. Faults in the two registers can malfunction the `load_store_unit` module.

In the last block of Table 3.8, a fault in the `branch_jump_set_done_q` affects executing multi-cycle branch and jump instructions. In the `load_stire_unit` module, the data needs to be extended. The `data_type_q` determines how the data is extended: word, half-word or byte extension. The `data_sign_ext_q` determines whether the data should be sign extended.

Some structures are less vulnerable and can only cause certain architectural property failures. For example, faults in most bits in the `instr_rdata_c_id_o` (which stores the compressed instruction) can only fail RVC properties; faults in the `data_type_q, data_sign_ext_q, data_we_q` in `load_store_unit` can only fail load and store properties. This finding can be used to partially protect vulnerable structures when the software executed in the core is known.

Strobe Properties There are 15 strobe signals, hence 15 strobe properties, as mentioned in subsection 3.3.1.2. The same method and tool were used to prove the strobe properties. Firstly we used Hts and Hps but got on average 690 undetermined results for each strobe property. Then AD, AM and case splitting were used to further analyse the undermined results. Finally, all the results of strobe properties were either 'Proven' or 'Failure'. We found that the results of all strobe properties are the same: if an SEU in a bit can/cannot corrupt one strobe, then it can/cannot corrupt all strobes.

The similarity is caused by the strong relationships among the strobes. Instructions in the core are important. Faults that alter the correct fetch/decode/execute process of an instruction can change values in all strobes. For example, if a fault alters the instruction and stalls the pipeline, `Instruction_is_done` fails. If a fault changes the source (rs1 and rs2) and destination (rd) register addresses, `Instruction`, `rs1_address`, `rs2_address` and `rd_address` can mismatch, which can further change `rs1_read_data`, `rs2_read_data` and `rd_write_data`. Similarly, if a fault alters a branch instruction, the PC may change; if a fault corrupts a load or store instruction, memory-related strobes may change.

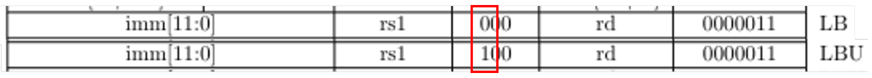
TABLE 3.9: Proven Results of Strobe Properties

Bits	Registers	Module
[4:11]	All except ctrl_fsm_cs	controller
[12:477]	All	CSR
[677:678]	imd_val_q	id_stage
683	instr_new_id_q	if_stage
[723:727],[731:740]	instr_rdata_alu_id_o	if_stage
749	instr_fetch_err_plus2_o	if_stage
[830:861]	addr_last_q	load_store_unit
[943:974]	stored_addr_q	prefetch_buffer
[975:1006]	fetch_addr_q	prefetch_buffer
1008	discard_req_q	prefetch_buffer

Table 3.9 lists the proven results (safe structures) with respect to all strobes. The other bits not in Table 3.9 are failure results (vulnerable structures). Proving strobe properties report 21270 crucial SEUs in 1418 bit locations. To save space, the vulnerable structures are not listed. The vulnerable bits identified by proving strobe properties cover those identified by proving architectural properties.

Comparison Strobe properties find bits vulnerable to strobes, while architectural properties find bits vulnerable to instructions. Proving strobe properties reports more crucial SEUs than proving architectural properties. We use strobe properties to complement architectural properties.

`rdata_q` in the `FIFO` stores instructions from the IMEM. `rdata_q` in the `load_store_unit` stores the data from the DMEM, `data_offset_q` and `handle_misaligned_q` in the `load_store_unit` determines the data extension and alignment. `valid_req_q`, `rdata_outstanding_q` and `branch_discard_q` play important roles in the control part in the `prefetch_buffer` module, faults in these bits can output wrong data to the core or stall the core. Faults in the `RegFile` can corrupt ALU results. A fault in the `core_busy_q` can stall the whole core. Faults in these bits/registers cannot be identified by architectural properties, because: 1) Faults in some structures (such as the `rdata_q` and `rdata_offset_q`) can only be identified by comparing them with golden references. For example, a fault in the `rdata_q` in the `FIFO` can change one valid instruction to another valid different instruction, such from RV32I_LB to RV32I_LBU, as shown in Figure 3.11. Such a fault cannot be identified by architectural properties because the retired instruction is RV32I_LBU instead of RV32I_LB. From the view of architectural proprieties, the retired instruction is valid and contains no fault. 2) Faults in some structures (such as `valid_req_q` and `core_busy_q`) refresh or stall the pipeline and stop instructions/data from propagation, the instructions/data cannot retire from the core and hence cannot be checked by architectural properties.



imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	100	rd	0000011	LBU

FIGURE 3.11: A fault changes RV32I_LB to RV32I_LBU

Though strobe properties produce a higher fault coverage, some of the identified SEUs are not really crucial SEUs. For example, if the faulty core produces correct results but takes more clock cycles than the golden core, then strobe properties fail because the strobes from the two cores are not equal. Such a mismatch can be reported as a counterexample, though it is not essentially an SDC.

In summary, architectural properties provide a narrow scope into instructions, but architectural properties cannot cover faults in all the structures in the core. Hence strobe properties are used to improve the fault coverage, though strobe properties cannot identify and classify faults based on instructions. By combining both properties, an exhaustive fault analysis of SDCs can be performed.

3.8.1.2 Crash

Table 3.10 shows the results of crash properties. There is no undetermined result. Hps and Hts were used as the engine modes. Most faults cannot cause a corresponding crash. The counterexamples were found after at least six clock cycles after reset: four clock cycles for the core to enter normal operation state and two clock cycles for fault propagation. `Insn_access_fault` represents a crash caused by an instruction access fault, such as invalid instruction address access. Similarly, `load_access_fault` and `store_access_fault` are crashes caused by invalid data address access. `Illegal_insn` represents a crash due to invalid instructions, such as instructions violating the RISC-V ISA specification.

With the help of COI analysis, there are 82 common structural safe bits, which are included in the Proven results. Compared to 81 safe bits that cannot cause an SDC, the extra safe bit is `instr_new_id_q`. This bit is used to capture strobe signals listed in Table 3.2. In the real Ibex Core, this bit is assigned to a value but never used. Although `instr_new_id_q` influences the signals used for formal verification, it does not influence the behaviour of the Ibex Core. Hence it is a safe bit.

TABLE 3.10: Bits vulnerable to crashes

Name	Proven	Failure
Insn_access_fault	2002	6
Illegal_insn	1908	100
breakpoint	1963	45
load_access_fault	2006	2
store_access_fault	2006	2
ECall_MMode	2004	4

Tables 3.11 to 3.15 list the crucial SEUs and vulnerable structures with respect to the six crash properties. Faults in some bits/registers can cause multiple crashes. For example, the bit 510 in the `valid_q` in the `FIFO` can cause three crashes: Insn_access_fault, breakpoint and ECall_MMode. Registers `valid_q` and `rdata_q` in the `FIFO` are the most vulnerable registers to crashes. Bits with faults causing multiple crashes are more vulnerable than bits with faults causing a single crash and hence need protection.

TABLE 3.11: Structures Vulnerable to Insn_access_fault

Bits	Register	Module
[608:610]	err_q	FIFO
748	instr_fetch_err_o	if_stage
[1013:1014]	rdata_pmp_err_q	prefetch_buffer

TABLE 3.12: Structures Vulnerable to Illegal_insn

Bits	Register	Module
478	instr_addr_q	FIFO
[509:511]	valid_q	FIFO
[512:607]	rdata_q	FIFO
[684:690],[696:698],[709:715]	instr_rdata_id_o	if_stage
767	illegal_c_insn_id_o	if_stage
[1009:1010]	rdata_outstanding_q	prefetch_buffer
2007	core_busy_o	Top

TABLE 3.13: Structures Vulnerable to breakpoint

Bits	Register	Module
478	instr_addr_q	FIFO
[509:511]	valid_q	FIFO
[512:523],526,[528:539],542,[544:555],558	rdata_q	FIFO
[560:571],574,[576:587],590,[592:603],606	rdata_q	FIFO
690	instr_rdata_id_o	if_stage
[1009:1010]	rdata_outstanding_q	prefetch_buffer
2007	core_busy_o	Top

TABLE 3.14: Structures Vulnerable to load_ and store_access_fault

Bits	Register	Module
866	pmp_err_q	load_store_unit
867	lsu_err_q	load_store_unit

TABLE 3.15: Structures Vulnerable to ECall_MMode

Bits	Register	Module
510	valid_q	FIFO
518,550,582	rdata_q	FIFO
690	instr_rdata_id_o	if_stage

3.8.1.3 Hang

The results show that faults in 1998 bits cannot cause a hang (WFI); faults in 10 bits can cause a hang (WFI). The identified crucial SEUs and vulnerable structures are listed in Table 3.16. `FIFO` is still the most vulnerable module and needs enhancement.

We found that software programs executed in the core play an important role when exploring faults causing hangs. The following is an example of the assembly code of a

TABLE 3.16: Structures Vulnerable to Hang (WFI)

Bits	Register	Module
478	instr_addr_q	FIFO
[509:510]	valid_q	FIFO
512,513,516,528,529,544,545,548	rdata_q	FIFO
560,561,576,577,580,592,593	rdata_q	FIFO
688	instr_rdata_id_o	if_stage
[1009:1010]	rdata_outstanding_q	prefetch_buffer
2007	core_busy_o	Top

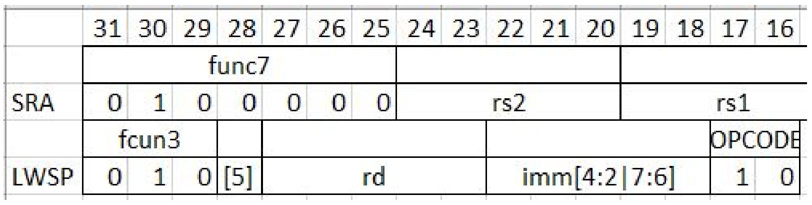
for-loop:

```
ADDI x1, x0, 10        # i = 10
loop:
   BEQZ x1, loopend    # if i == 0, break loop
   SUB x1, x1, 1       # i = i - 1
   JAl x0, loop        # jump to loop
loopend:
   ADDI x3, x1, 0      # a = i
```

In the above assembly code, if the value in register `x1` is zero, the for-loop ends with the instruction `BEQZ`. When `x1` is zero, if there is a fault when reading or executing `BEQZ`, then the for-loop will never end, causing a hang. Several faults can cause such a problem. For example, faults causing an incorrect instruction address such that `BEQZ` is either not fetched from `FIFO` or read from memory; or wrong results (faults in `RegFile`) of `BEQZ`. Apart from faults in hardware, we found faults in software (such as the above example) can lead to a hang. To find all faults causing hangs, both software and hardware need to be considered. It might be easier to find faults causing hangs with hardware running a known software program. A known software program constrains the input state space, hence a great amount of computation effort exploring scenarios outside the given software program can be reduced.

3.8.2 Instruction-level

The value in the `instr_is_compressed_id_o` in the `if_stage` determines whether the instruction is 32-bit or 16-bit. We found that all RV32I instructions are vulnerable to this register because a fault in this register can force the core to treat RV32I instructions as RVC instructions. In most cases, illegal RVC instructions are generated and a crash is reported. The only exception is the RV32I_SRA instruction because the most significant 16 bits can be treated as a valid RVC_LWSP instruction, as shown in Figure 3.12. To save space, only the left 16 bits in the RV32I_SRA are displayed. The left 16 bits match the RVC_LWSP format: the fun3 code is 010 and the OPCODE is 10 if bit 17 is 1 and bit 16 is 0 in the RV32I_SRA. On the contrary, all RVC instructions are reliable to the `instr_is_compressed_id_o` because two combined RVC instructions can be treated as a valid RV32I instruction.



	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	func7															
SRA	0	1	0	0	0	0	0	rs2					rs1			
	fcun3														OPCODE	
LWSP	0	1	0	[5]	rd					imm[4:2|7:6]					1	0

FIGURE 3.12: The left 16 bits of RV32I_SRA can be decoded as RVC_LWSP

Instructions that perform similar functions in different instruction sets are placed on the same row in Table 3.6. For convenience, we extract those instructions to Table 3.17. It can be seen that the compressed instructions are more vulnerable than the base instructions. Hence, implementing fewer or no RVC instructions can improve the reliability of a system in the presence of soft errors.

TABLE 3.17: RV32IC instructions that perform similar functions

Name	Failure	Proven	Name	Failure	Proven
RV32I_LUI	91	1917	RVC_LUI	130	1878
RV32I_ADD	113	1895	RVC_ADD	126	1882
RV32I_ADDI	112	1896	RVC_ADDI	129	1879
RV32I_XOR	113	1895	RVC_XOR	128	1880
RV32I_OR	112	1896	RVC_OR	128	1880
RV32I_AND	112	1896	RVC_AND	126	1882
RV32I_ANDI	112	1896	RVC_ANDI	130	1878
RV32I_SLLI	113	1895	RVC_SLLI	127	1881
RV32I_SRLI	118	1890	RVC_SRLI	131	1877
RV32I_SUB	107	1901	RVC_SUB	129	1879
RV32I_JAL	121	1887	RVC_JAL	134	1874
RV32I_JALR	81	1927	RVC_JALR	90	1918
RV32I_BEQ	114	1894	RVC_BEQZ	130	1878
RV32I_BNE	107	1901	RVC_BNEZ	121	1887
RV32I_LW	106	1902	RVC_LW	133	1875
RV32I_SW	106	1902	RVC_SW	131	1877

We group RV32IMC instructions into seven groups: RV32I and RVC Integer Computational Instructions, RV32I and RVC Control Transfer Instructions, RV32I and RVC Load and Store Instructions and RV32M Instructions. We found that the instructions are reliable for some common structures, as listed in Table 3.7. We also found different groups of instructions have different vulnerabilities. For example, all RV32I instructions are vulnerable to all four bits in the `ctrl_fsm_cs` in the `controller`, while RVC instructions are only vulnerable to the least significant two bits, and RV32M instructions are vulnerable to the least significant three bits. Only

RV32M instructions are vulnerable to multiplier-related structures, such as the `imd_val_q` register and the `MULTDIV` module. Only RV32IC Control Transfer Instructions are vulnerable to SEUs in the `branch_jump_set_done_q` which indicates the complement of a branch/jump. The RV32IC Integer Computational Instructions are more vulnerable to the `instr_rdata_alu_id_o` in the `if_stage` than others. The RV32IC Load and Store Instructions are more vulnerable to the `load_and_store_unit` than others. If the vulnerable hardware is not protected, then corresponding instructions should be enhanced to improve the system's reliability.

3.9 Discussion

The proposed method can successfully evaluate the hardware reliability in the presence of SEUs. We use formal verification to explore the entire state space and the whole fault list to search all crucial SEUs that can cause SDCs/crashes/hangs, and all safe SEUs that cannot cause SDCs/crashes/hangs. It is found that some bits are more vulnerable to SEUs than others. For example, the 1-bit register `instr_valid_id_q` is very vulnerable because faults in this bit can cause all types of SDCs. In addition, some registers are only vulnerable to certain failures. For example, the register `ctrl_fsm_cs` is only vulnerable to SDCs.

We found that some faults can cause multiple (property assertion) failures. For instance, all the 31 bits in the register `instr_addr_q` in the `FIFO` module can violate all architectural and strobe properties. In other words, faults in these bits may cause all the failures modelled by the architectural and strobe properties. Hence, these bits are the most vulnerable bits and must be protected to minimize SDCs. In addition, `pc_id_o`, the program counter, is a very vulnerable register. Most bits in the program counter can fail all the assertions and hence need protection. It is important to protect the `FIFO` and the PC with fault-tolerant technologies.

We also found that faults in both the data path and control path can lead to SDCs. For instance, faults in the register `ctrl_fsm_cs` in the `controller` (in the control path) can lead to SDCs; faults in the `RegFile` (in the data path) can also lead to SDCs.

Not all bits in the same register are vulnerable to the same failure. For example, in Table 3.8, the `instr_rdata_alu_id_o` register contains 32 bits; some bits can cause more than 50 failures while some bits can cause less than 20 failures. Another example is in Table 3.12, not all bits in the `instr_rdata_id_o` register can cause an illegal_insn crash. As a result, partially protecting the most vulnerable bits instead of the entire registers could be a cost-efficient method to improve hardware reliability.

Formally verifying RV32IC Integer Computational Instructions is easy because they are single clock cycle instructions. It is harder to formally verify RV32IC Control

Transfer Instructions, because branches and jumps require multiple clock cycles. Formally verifying RV32IRVC Load and Store Instructions is the most difficult part, because the `load_store_unit` is involved which increases the verification complexity. Fully proven results can be reported for the above instruction sets with our complexity control strategies. However, we failed to eliminate the undetermined results of RV32M Instructions because of the `MULTDIV` module.

The `MULTDIV` module contains a multiplier and a divider. Formally verifying the `MULTDIV` module is the most challenging part of this research. We tried to abstract the `MULTDIV` module to reduce verification complexity. There are three parallel 17x17 multipliers in the `MULTDIV` module. The following is a property that models the RV32M_MUL instruction. A 32x32 multiplication is performed by the property to calculate the theoretical multiplication result.

```
logic [63:0] mul_res;
assign mul_res =
$signed($signed(rs1_rdata) * $signed(rs2_rdata));
property p_RV32M_MUL;
  rst_ni && RV32M_MUL |-> (rd_wdata == ((rd_addr == ‘x0)
  ? ’0 : mul_res[31:0])) && (pc_wdata == pc_rdata+4);
endproperty
```

The three multipliers in the `MULTDIV` cannot be simply black-boxed, because with black-boxing FPV can ignore the real multiplication process and try to construct a multiplication result that violates corresponding RV32M properties. For example, if both operands of a MUL are 0, the theoretical result should be 0. During model checking, FPV ignores the multiplication process and derives the result as 2. Hence a counterexample is reported. This result is meaningless because we cannot find any crucial SEUs from the counterexample.

What is worse, we found that FPV black-boxed all multipliers, even in the RM32M architectural properties. In other words, the $*$ operator in the third line of the above example is also black-boxed. We could not correctly model the `MULTIDV`, so we could not get correct theoretical multiplication/division results. As a result, we failed to apply black-boxing to fully prove the RV32M properties in the presence of SEUs. On the other hand, this chapter aims to demonstrate our method. One key point of our research is to construct and formulate different properties which reveal SEUs that lead to failures such as SDCs, crashes and hangs. Completely verifying multipliers and dividers with formal verification is not the objective of our research. As a result, we did not solve undetermined results with RV32M instructions.

Apart from hardware, one notable finding is that an inappropriate sequence of instructions (misaligned instructions) can lead to system failures. For example, the jump address of branch instructions must be aligned by four. Otherwise, the Ibex core will try to align the instruction stored in the jump address by reversing the upper 16 bits with the lower 16 bits. In some cases, reversed instructions are illegal, causing illegal_insn exceptions even without any faults. Similar problems exist in other properties, such as the for-loop example in the last section. This kind of problem should never arise with a good compiler. In addition, in the presence of SEUs, the RV32I instructions are more reliable than the RVC instructions, hence it is better not to implement RVC instructions in safety-critical designs.

Faults in one bit may cause multiple types of errors. The more types of errors a fault may cause, the more vulnerable the bit is. Thus the reliability of all bits can be ranked. In general, most of the bits in the Ibex Core are vulnerable to SDCs, and the second pipeline stage is more vulnerable to SDCs than the first pipeline stage. As a result, protecting the bits in the second pipeline stage could be a cost-efficient way to mitigate SDCs. On the other hand, registers that store instructions and PC are the most vulnerable structures. Protecting the `FIFO` and PC is the most important task in safety-critical designs.

There are differences between the proposed method and the reviewed works in Chapter 2. The first difference is that we do not need to specify the activation condition for each fault. Jayakumar manually models state conditions that indicate an error or a system failure as assumptions and uses model checking to identify fault activation conditions [1]. Then he uses these conditions to activate faults to verify safety properties. It is difficult to manually model such assumptions. In addition, Jayakumar's work is verifying injected faults with safety properties. Our proposed work is different. We assume a fault could occur in each bit at each time. We model properties according to fault effects and use model checking to find counterexamples indicating faults causing system failures. It is unnecessary to specify fault activation conditions since once a counterexample is found, the corresponding activation condition is reported by the formal tool.

The second difference is that our properties are general to all RISC-V processors. Other works are either application-specific [1] or software-specific [140]. In other words, our method can be used to evaluate hardware reliability in other RISC-V designs.

The third difference is that our method is friendly to hardware designers and verifiers. Samadi's method involves creating a library to transform the DUT to FT models [121]. It is hard to create such a library. In our method, the DUT is at the RTL level, which is general in hardware design.

The fourth difference is that we developed input assumptions/constraints to avoid false negatives. Most works leave inputs to DUT unconstrained. We found that unconstrained inputs may lead to false negatives: failure may arise with unconstrained inputs and no faults. The details are in Section 3.4. The false negative may decrease the result accuracy. As a result, we developed a set of assumptions to constrain inputs to avoid false negatives.

The last and most important difference is that we identified and categorized faults according to fault effects. Other formal works just classify faults into two groups: crucial faults and safe faults. Our method can find both. In addition, our method can further categorize faults according to fault effects. Our developed properties can reveal faults that lead to different failures. Our method can find all faults causing nothing, and faults causing SDCs, crashes and hangs.

It is theoretically possible but practically impossible to use simulations to explore all SEUs. However, we have successfully explored all SEUs using our formal method. It might be argued that technologies, such as random fault injection and fault pruning, and high-performance computing hardware can be applied to speed up simulation. We argue that with the same time and computing hardware, formal methods can explore and identify more crucial faults than simulation-based approaches. In addition, our method is more exhaustive than simulation. It is hard to cover all the state space with simulation, as stimuli must be chosen carefully to reach corner states. The results of simulation-based fault injection are program-dependent. To get fair results, more programs are needed, needing more time and effort. On the other hand, our method is program-independent and can explore all the state space. It is hard and tedious to use simulation to find all faults that can cause a given error. Our method can identify all crucial faults for a given error, which meets the objective of this research.

There is one limitation to the current work. Architectural properties cannot cover all bit locations in the Ibex Core. Though we used strobe properties to complement the fault coverage of architectural properties, strobe properties can report the scenario 'the faulty core produces correct results but takes more clock cycles than the golden core' as SDCs, which is a false negative.

3.10 Conclusion

One major concern of embedded systems is Single Event Upsets. Single Event Upsets, also called soft errors, can cause system failures or can have no effect. This chapter proposes a formal method that covers the whole state space and the whole fault list and performs formal fault analysis to identify and classify crucial SEUs that lead to SDCs, crashes and hangs. The main contribution of this work is not using formal verification to perform fault analysis, but developing formal properties to identify faults that lead to SDCs, crashes and hangs. We demonstrate the method on a RISC-V Ibex Core. The results show that both the hardware (bits/registers/modules) reliability and the software (instructions) reliability can be evaluated. Compared to simulation-based fault injection, the advantages of this method are exhaustive search and short runtime. Due to the advantages, we will expand the method to evaluate fault-tolerant technologies and Double Event Upsets in the next two Chapters.

Chapter 4

Evaluating Fault Tolerant Technologies

In Chapter 3 we developed properties for revealing faults that lead to failures (SDCs, crashes and hangs), and applied our formal method to evaluate the hardware reliability of a RISC-V Ibex Core in the presence of SEUs. We used that method to identify vulnerable bits to SEUs. We found some bits are more vulnerable than others, and some faults are more critical (may cause more serious failures) than others. This chapter extends the previous work to evaluate fault-tolerant technologies.

Fault-tolerant technologies can be used to enhance hardware reliability against soft errors. Different fault-tolerant technologies have different fault detection and protection effectiveness and different overheads. It is more cost-effective to protect the most vulnerable structures with the most effective (but the most expensive) technology, and less vulnerable structures with less effective (and cheaper) technologies, and leave reliable structures untouched. We aim to extend our formal method in Chapter 3 to find all SEUs that might cause errors/system failures under the protection of fault-tolerant technologies. The identified SEUs can be used to evaluate the detection effectiveness of the implemented fault-tolerant technologies. The evaluation results help to determine cost-efficient fault-tolerant technologies.

In Section 3.7 we applied our method to evaluate two well-known fault-tolerant technologies: Triple Modular Redundancy (TMR) and Shadow Registers (SR). The results were used to validate our method. In this chapter, we demonstrate how to apply our method to evaluate the effectiveness of residue arithmetic. Compared to simulation-based methods, our method can test all faults and cover all the state space in a reasonable time and is independent of testbenches. A script is used to automatically carry out the entire experiment process. The developed method is compatible with other RISC-V processors, because the developed method contains all the formal properties and constraints compatible with verifying other RISC-V processors. In addition, the proposed method is easy to use, and users require no formal knowledge. The only alteration needed to adapt the method to other designs is to map signal names.

4.1 Method Overview

The basic idea of the method in this chapter is extending the formal method in Chapter 3 to identify: 1) all hardware structures (bits, registers and modules) where injected faults (whether safe or crucial) can be detected by the implemented fault-tolerant technologies; and 2) all crucial faults that can lead to failures (SDCs, crashes and hangs) even in the presence of fault-tolerant technologies. The extended method that evaluates the effectiveness of fault-tolerant technologies in this chapter can be divided into five steps, as shown in the following:

1. Implement a fault-tolerant technology into the Ibex Core.
2. Implement a fault injection mechanism.
3. Develop formal properties that can evaluate raw and crucial fault detection effectiveness of the implemented fault-tolerant technology.
4. Develop complexity control strategies to improve proof performance.
5. Perform model checking to evaluate the implemented fault-tolerant technology in the presence of SEUs.

The first step is to implement residue arithmetic into the Ibex Core. The details of residue arithmetic, including its principle and how it is implemented into the Ibex Core will be provided in Section 4.2.

The second step is to implement the fault injection mechanism described in Section 3.2 into the Ibex Core. Residue arithmetic adds extra residue bits into the Ibex Core. Faults in the extra residue bits are also considered. Hence, the fault injection mechanism is slightly modified so that faults can be injected into both the original bit locations and the extra residue bit locations. Since the modification is simple, the details of the fault injection mechanism will not be repeated in this chapter.

The third step is to develop formal properties to reveal all faults that can be detected by fault-tolerant technologies. The detected faults may or may not cause failures. The properties in Section 3.3 are then modified to identify faults that lead to SDCs, crashes and hangs. The details of the new developed properties and the modified properties will be given in Section 4.3.

In the fourth step the same complexity control strategies in Section 3.4 are used in this Chapter.

The final step is to perform model checking. The experimental strategy and the formal tool settings are slightly different from those in Section 3.5. The details will be given in Section 4.4.

Similar to Chapter 3, the formal properties are written as SystemVerilog Assertions (SVAs). Cadence JasperGold FPV is used to perform model checking. The Ibex Core is chosen as the test platform. An SEU is chosen as the fault model. Proven, Undetermined (Bounded Proven), and Failure are used to describe model checking results.

4.2 Residue Arithmetic

Residue arithmetic detects single faults that may result in errors and system failures caused by arithmetic computation errors. For example, a microprocessor executes an addition operation: $Sum = OP1 + OP2$. In a residue number system, equation (4.1) applies. m is a prime number no less than 3. % is the modulus operator.

$$Sum \,\%\, m = (OP1 \,\%\, m + OP2 \,\%\, m) \,\%\, m \tag{4.1}$$

With the help of the above equation, the correctness of the sum results can be checked. Besides addition, residue arithmetic can be applied to subtraction, multiplication, and division. Hence, residue arithmetic can be implemented in hardware to detect arithmetic errors.

Residue arithmetic itself can only detect faults; it cannot correct faults. Extra fault correction technologies are required to correct the detected faults. We assume extra fault correction technologies will be implemented after the analysis, hence all the detected faults will be corrected. Such an assumption can be used to evaluate the effectiveness of residue arithmetic, the details are in Section 4.3.

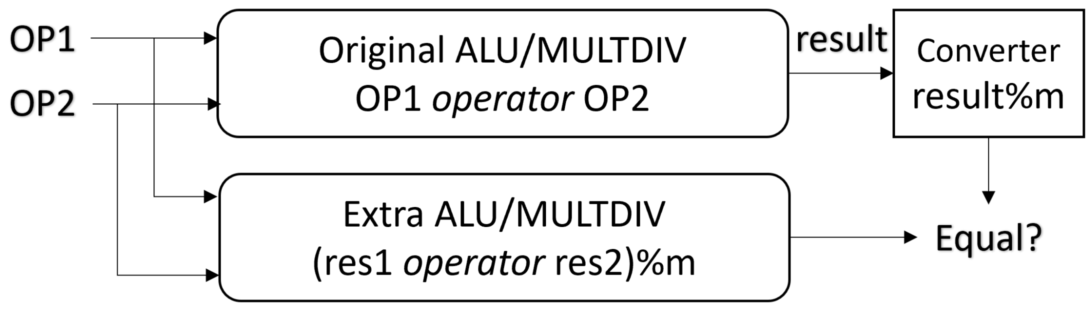


FIGURE 4.1: Block Diagram of Residue Arithmetic in Ibex

Figure 4.1 shows the implementation of residue arithmetic in the Ibex Core. The original ALU and MULTDIV perform normal 32-bit arithmetic operations. A residue converter is added at the end to compute the residue of the normal result (LHS in equation (4.1)). An extra ALU and MULTDIV are added to perform residue calculation (RHS in equation (4.1)).

Inputs to the extra ALU and MULTDIV are residues rather than 32-bit integers. In order to minimize hardware and computation overhead, converters that compute input residues are not implemented before the extra ALU and MULTDIV. In other words, input residues are not computed every time an arithmetic operation is executed. Instead, we add extra residue bits to corresponding registers.

The residue of a number is less than its modulus. To minimize the hardware overhead, the smallest possible modulus, 3, is chosen. As a result, two extra residue bits would be added to each register in the Ibex Core. Other modulus are greater than 3 and will increase the number of extra residue bits.

After reset, all the residue bits are zero. The residue bits are computed by the residue converter after the original ALU and MULTDIV. The extra ALU and MULTDIV perform the following operation:

$OP1\_residue = OP1\%m, OP2\_residue = OP2\%m.$

If equal, Figure 4.1, the resulting residue will be stored in the residue bits in the destination register. By doing so extra computations can be avoided.

Residue arithmetic adds extra residue bits. Faults in the extra residue bits are also considered. A residue converter before the extra ALU and MULTDIV cannot detect faults in the operands due to topology. For example, the correct operation in the original ALU and the converter is

$(5 + 3)\%3 = 2;$

The correct operation in the extra ALU is

$(5\%3 + 3\%3)\%3 = 2.$

If a fault occurs in the Least Significant Bit in the first operand, the first operand 5 becomes 4; the operation in the original ALU and the converter becomes:

$(4 + 3)\%3 = 1;$

The operation in the residue ALU becomes

$(4\%3 + 3\%3)\%3 = 1.$

The fault is hidden by the converter, so the fault can never be detected by residue arithmetic.

Therefore, we implement residue bits to all registers that can directly affect the arithmetic operations through the Ibex Core. We perform backward tracing of these registers. Both COI analysis and static slicing [54] approaches are involved in this process. Details of COI analysis and static slicing are in Section 3.5.1 and Appendix G. Only registers in the Register File are added with residue bits. We will explain the reasons in the following paragraphs.

Residue arithmetic focuses on comparing residues of arithmetic operations, as shown in Figure 4.1. To reduce computation efforts, the residues are stored in the corresponding registers, so that it is unnecessary to compute residues of ALU/MULTDIV inputs frequently. To reduce the hardware overhead, the extra residue bits are only added to source registers storing input data to ALU/MULTDIV and destination registers storing output data from ALU/MULTDIV. In other words, adding extra residue bits to the Register File is enough to cover all residues involved in Figure 4.1.

It might be argued that faults in other structures, such as the control path, may alter arithmetic operations. For example, a fault in the register storing the fetched instruction (such as the `rdata_q` in the `FIFO`) may change an addition operation to others, hence causing an incorrect arithmetic result. However, though such a fault may lead to errors or system failures, this fault cannot be detected by residue arithmetic. As shown in Figure 4.1, residue arithmetic cannot validate whether the operator is correct or not. We have discussed such faults in Chapter 3. In addition, investigating this argument, such as the impacts of faults outside the data path, is an objective of the experiment. Results will be given in Section 4.5.

In addition, it might be argued that registers in ALU/MULTDIV are directly involved in arithmetic operation, hence it is necessary to add residue bits to enhance them. The ALU in the Ibex Core is purely combinational logic and there are no registers inside it. Only registers in the Register File can directly affect the ALU arithmetic results. For example, source registers storing $OP1$ and $OP2$ in Figure 4.1. Sometimes, $OP2$ can be an immediate value extracted from the instruction. The extraction process is also purely combinational logic in the Ibex Core. The residue of the immediate value can be computed by adding an extra residue converter. However, faults in the instruction corrupting the immediate value cannot be detected by residue arithmetic, because the (faulty) immediate value is directly fed into the extra residue converter. Extending instructions with extra residue bits may solve the problem. However, extending the instruction size requires adding extra residue bits to almost all registers in the Ibex Core, which is too expensive. As a result, residue bits are only added to the Register File.

The MULTDIV module in the Ibex Core is a state machine to perform multiplication and division. As stated in the Ibex Reference Guide, a Fast Multi-Cycle Multiplier is

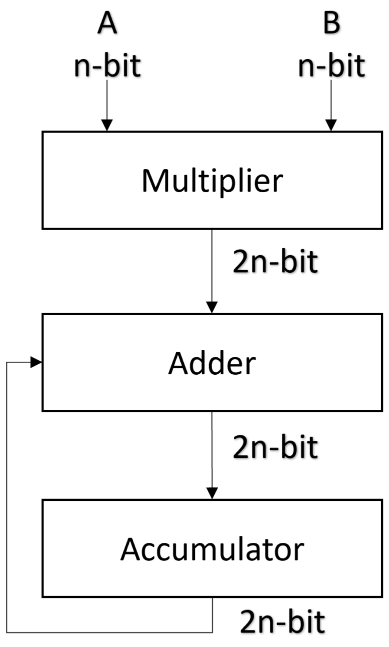


FIGURE 4.2: Block Diagram of MAC

implemented, because it provides a reasonable trade-off between area and performance, and it is the first choice for ASIC synthesis [26]. The multiplier completes multiplication in three to four clock cycles: a MUL instruction takes three clock cycles, and a MULH instruction takes four clock cycles. The Multiply-accumulate (MAC) operation is implemented in the multiplier. A basic MAC architecture consists of the formation of partial products and an accumulative addition [141]. As shown in Figure 4.2, a typical n-bit MAC unit contains an n-bit multiplier, a 2n-bit adder, and a 2n-bit accumulator. ALU is used to perform addition. In the Ibex Core, in order to reduce the size of the MAC, the multiplier divides a 32-bit number into two 16-bit numbers and adds a sign-extension bit to each. As a result, the MAC is capable of a 17-bit x 17-bit multiplication with a 34-bit accumulator. However, the MAC is still implemented with the $*$ and $+$ operators. As discussed in Section 3.9, we cannot black box multipliers to reduce the verification complexity.

Though there are registers in the MAC, it is unnecessary to add residue bits to them. Adding residue bits to the MAC registers brings another residue operation (an extra MAC, a converter, and a comparator), which is expensive. In addition, if a fault is injected into the MAC registers and causes a wrong result, the existing residue arithmetic in Figure 4.1 can detect it. As a result, it is unnecessary to add residue bits to the registers inside the MULTDIV. Since the extra MULTDIV performs multiplication on two-bit residues, and in order to make it simple, we choose to implement the extra MULTDIV without a MAC, as shown in Appendix H.

4.3 Formal Properties

Developing appropriate formal properties is important in formal verification. Based on the basic method idea in Section 5.1, we developed two groups of formal properties. The first group is named Raw Fault Detection properties, which aim to identify all injected faults that can be detected by the implemented fault-tolerant technologies. The second group, which is named Crucial Fault Detection properties, is a modified version of the properties in Section 3.3. Crucial Fault Detection properties identify faults that can lead to failures (SDCs, crashes and hangs) even in the presence of fault-tolerant technologies. The first group evaluates the raw fault detection effectiveness. The second group evaluates the crucial fault detection effectiveness. In the following section, we first introduce the first group of properties, and then we introduce the second group of properties.

4.3.1 Raw Fault Detection

Two extra modules, an ALU and a MULTDIV, are added for residue arithmetic. Each module has a unique error indication signal. If a single fault is injected, causing different residue results between the original and the extra ALU, the ALU error indication signal will be set by the comparator in Figure 4.1. This is the same as MULTDIV. As a result, two raw fault detection properties are developed to monitor the two error indication signals, as shown in the following.

```
property p_alu_ra_error;
  !alu_ra_error;
endproperty

property p_mul_ra_error;
  !mul_ra_error;
endproperty
```

The first property `p_alu_ra_error` specifies that the ALU error indication signal `alu_ra_error` should never be set. An assertion failure means an injected fault has been detected by the residue arithmetic protecting the ALU. Similarly, the second property `p_mul_ra_error` checks whether an injected fault can be detected by residue arithmetic in the MULTDIV, by monitoring the MULTDIV error indication signal `mul_ra_error`.

The above two properties are used to evaluate the raw fault detection effectiveness of residue arithmetic. For example, if faults in all bit locations in the Ibex Core could violate the first property assertion, which means faults in all bits could be detected by the ALU residue arithmetic, then 100% raw fault detection efficiency is achieved by the ALU residue arithmetic. This is the same as the MULTDIV residue arithmetic. On the other hand, If both assertions were proven independent of fault attributes, then the implemented residue arithmetic cannot detect any faults in the Ibex Core.

4.3.2 Crucial Fault Detection

In this subsection, all the properties are modified from the properties in Section 3.3. We use the modified properties to reveal faults that can be detected and lead to SDCs, crashes and hangs. In the following, we only introduce how to modify these properties, and how to use them to evaluate crucial fault detection effectiveness.

4.3.2.1 SDC

We choose ISA-independent strobe properties to identify faults that can cause SDCs in the protection of residue arithmetic. One main reason for using strobe properties instead of architectural properties is that strobe properties can cover the Register File, where extra residue bits are added. Detailed comparisons of strobe properties and architectural properties including fault coverage have been given in Chapter 3. Strobe properties duplicate the Ibex Core. Different from the last chapter, both cores are protected by residue arithmetic. Another difference is that the two error indication signals `alu_ra_error` and `mul_ra_error` are added to strobe properties as antecedents respectively. The following are two SDC assertion examples modified from Section 3.3.

```
a_rd_wdata_alu: assert property (
 golden_valid && faulty_valid && alu_ra_error|->
 faulty_rd_wdata == golden_rd_wdata );

a_rd_wdata_multdiv: assert property (
 golden_valid && faulty_valid && mul_ra_error|->
 faulty_rd_wdata == golden_rd_wdata );
```

The above two assertions specify that if both retired instructions from the two cores are valid RM32IMC instructions (`golden_rvfi_valid` and `faulty_rvfi_valid` set at the same time), and the residue arithmetic implemented in the ALU/MULTDIV detects a fault respectively (`alu_ra_error` set or `mul_ra_error` set), then the write data to the register file from the two cores (`golden_rd_wdata` and `faulty_rd_wdata`) should be the same.

Failure of the first assertion reveals a fault that has been detected by the residue arithmetic in ALU can corrupt write data to the register file, hence causing an SDC.

This is similar to the second assertion except in MULTDIV. If the above assertions were proven, all faults that can be detected by the residue arithmetic in ALU/MULTDIV bits cannot cause failures. We have found all crucial SEUs in the Ibex Core without any fault-tolerant technologies in Section 3.8. By comparing both results, crucial SEUs that cannot be detected and mitigated by the residue arithmetic in ALU/MULTDIV can be found. Hence the crucial fault detection effectiveness can be calculated. Details will be given in Section 4.5.

4.3.2.2 Crash

Similar to modifying the SDC properties as above, we modified the crash properties in Section 3.3 to evaluate residue arithmetic. Error indication signals are added as antecedents. The following are two examples of modified crash properties.

```
a_store_access_fault_alu:
assert property (
 (crash_priv_mode==2'b11) && alu_ra_error |->
 (crash_mcause_q!=6'd7) );

a_store_access_fault_multdiv:
assert property (
 (crash_priv_mode==2'b11) && mul_ra_error |->
 (crash_mcause_q!=6'd7) );
```

The first assertion *a_store_access_fault_alu* specifies that, when the core is operating in the machine mode, `crash_priv_mode==2'b11`, if a fault has been detected by the residue arithmetic in ALU, the error indication signal `alu_ra_error` is set to logic high, then the detected fault should never cause a store access fault crash, with exception code 7 in the CSR register `mcause`. If the first assertion fails, a crucial fault that can result in a (store access fault) crash and can be detected by the residue arithmetic in ALU has been identified. We assume extra fault correction technologies will be implemented to correct all the detected crucial faults after the analysis. Hence all the detected and crucial SEUs identified from the assertion failures will be corrected. If the first assertion was proven without a vacuous pass (the current privilege mode is the machine mode and the ALU residue arithmetic can detect injected faults), then all the detected faults can cause no crash. The second assertion *a_store_access_fault_multdiv* is similar except the error indication signal `mul_ra_error` is from the residue arithmetic in MULTDIV.

4.3.2.3 Hang

In order to avoid hard-to-solve problems such as liveness properties and state explosion, only the case where 'the core is sleeping and waiting for a new instruction' is modelled as a formal property. The following are the corresponding assertions of the residue arithmetic in ALU and MULTDIV respectively. The error indication signals are added as antecedents separately.

```
a_hang_alu:
assert property (
valid && !halt && alu_ra_error |->
(insn != 32'h10500073) );
a_hang_multdiv:
assert property (
valid && !halt && mul_ra_error |->
(insn != 32'h10500073) );
```

The above two assertions are similar. The only difference is the source of the error indication signal. The above assertions specify that any detected faults by the residue arithmetic in ALU/MULTDIV should never cause hangs. Signals (except the error indication signals) involved in the above two assertions have been explained in Section 3.3.

If the first assertion fails, a detected crucial fault by the ALU residue arithmetic that can result in a hang has been identified. We assume after the analysis, extra fault correction technologies will be implemented to prevent detected crucial faults from propagating and, hence cannot cause hangs. In other words, the identified crucial faults can be mitigated by the ALU residue arithmetic. If the first assertion was proven without a vacuous pass, then all the detected faults in the ALU residue arithmetic cannot cause hangs.

4.4 Model Checking

We perform model checking with the above properties to analyse faults. If an assertion fails, a CEX (including the fault location and time) will be given, thus the fault can be extracted from the CEX. In this section, we first introduce our experimental strategy, and then we provide the formal tool settings.

4.4.1 Experimental Strategy

We first perform the experimental strategy in subsection 3.5.1 (without any changes) to prove the raw fault detection properties in subsection 4.3.1. This step reports all detected faults, hence all detectable bit locations by the ALU residue arithmetic and the MULTDIV residue arithmetic.

Then we modify the experimental strategy to prove the crucial fault detection properties in subsection 4.3.2. In step a) in subsection 3.5.1 the SEU is no longer arbitrary in the whole fault space; only faults in the detectable bit locations are explored. It is meaningless to explore faults in undetectable bit locations, as error indication signals will not set, and proving the crucial fault detection properties in subsection 4.3.2 can only report vacuous pass. Removing undetectable bit locations from the fault space greatly reduces the time and effort for fault analysis. The other steps of the experimental strategy are the same as those in subsection 3.5.1, hence they are not repeated in this subsection.

4.4.2 Configurations of Cadence JasperGold

In general, the tool settings are the same as the previous settings in subsection 3.5.2. The proof time was limited to five hours because, in the last chapter, we found some cases took up to five hours to be fully proved. We did not limit the proof trace. We enabled ProofMaster to improve proof performance. We set the number of max jobs per model checking run to 10 in case of security and ran two FPVs in parallel. One thread was assigned to each engine because each core in the server only has one thread.

The same engine modes in subsection 3.5.2 are used to prove the properties in Section 4.3. The detailed engines are shown in Table 4.1. The second and the third columns represent the residue arithmetic in the ALU and the MULTDIV respectively. The second and third rows list the engines that falsified and fully proved the raw fault detection properties in subsection 4.3.1. Similarly, the last two rows list the engines used to find counterexamples and prove the crucial fault detection properties in subsection 4.3.2.

TABLE 4.1: Engines used to prove properties in Section 4.3

	ALU RA	MULTDIV RA
Raw CEX	Hts	Hts AD
Raw Proven	Hps	Hps...AM
Crucial CEX	Hts	Hts AD
Crucial Proven	Hps	Hps...AM

It was easiest to disprove and prove the ALU residue arithmetic with the raw fault detection properties, as Hts and Hps can solve all the problems. It was harder to fully prove the ALU residue arithmetic with the crucial fault detection properties, and case splitting was used to improve the performance of Hps. Nevertheless, the ALU residue arithmetic can be fully proved with our complexity control strategies.

On the other hand, we could not fully prove the MULTDIV residue arithmetic, because of the multiplier and the divider in the MULTDIV, as discussed in Chapter 3. We used Hts and AD with case splitting to find counterexamples of raw and crucial fault detection properties with the MULTDIV residue arithmetic. We tried different engines such as Hps, N, R and AM but only got undetermined results after five hours.

4.5 Results

There are three outcomes from model checking. 'Failure' means the assertion fails, i.e. the detected fault can cause a system failure modelled by the assertion. 'Proven' means the assertion is fully proven; all the detected faults cannot cause system failures. 'Undetermined' means that due to time limit or state explosion, it is too difficult to prove or disprove the assertion, so 'Bounded Proven' is reported. In this chapter, all the undetermined results are reported from proving raw and crucial fault detection properties with the MULTDIV residue arithmetic. In other words, the ALU residue arithmetic is fully proved.

We applied the same complexity control strategies in Section 3.4, such as constraining the input instructions as aligned and valid RV32IMC instructions. However, we removed the RV32M instructions from the input assumptions when we evaluated the residue arithmetic enhanced (RAE) ALU. The MULTDIV is used to perform multiplications and divisions. ALU is used by the MULTDIV for accumulations. However, during experiments, we found that evaluating the RAE ALU with RV32M instructions took a long time and sometimes the properties could not be fully proved, because the MULTDIV is involved by RV32M instructions. Hence we excluded the RV32M instructions from the input space only when evaluating the RAE ALU. The verification complexity was greatly reduced without computing the MULTDIV. This is why there are no undetermined results for the RAE ALU. On the other hand, we included the RV32M instructions to evaluate the RAE MULTDIV, otherwise the MULTDIV could not be triggered. It only took minutes to disprove each property of the ALU residue arithmetic. 90% properties of the ALU residue arithmetic can be fully proved within one hour, and the remaining properties took up to five hours. It also took minutes to disprove the MULTIDIV residue arithmetic, but the MULTIDIV could not be fully proved.

Some fault-tolerant technologies, such as residue arithmetic, can only detect SEUs in certain bit locations. We call the number of detectable bit locations to the number of total bit locations the raw detection effectiveness. The faults injected in the detectable bit locations and can be detected are detected faults. Not each detected fault can cause failures. Detected crucial faults are the detected faults that cause failures. Detected crucial faults can be corrected by other technologies such as software redundancy.

By proving the raw fault detection assertions in subsection 4.3.1, the detectable bit locations can be obtained. The raw fault detection effectiveness of fault-tolerant technologies can be calculated as:

$$DetectableBitLocations \div TotalBitLocations \tag{4.2}$$

The number of total bit locations varies because fault-tolerant technologies add extra bits to the Ibex Core.

By proving the crucial fault detection assertions, such as the modified strobe, crash, and hang assertions in subsection 4.3.2, the detected crucial faults can be identified. The crucial fault detection effectiveness of fault-tolerant technologies can be calculated as:

$$DetectedCrucialFaults \div TotalCrucialFaults \tag{4.3}$$

The total crucial faults (without any fault-tolerant technologies) have been identified in Chapter 3.

In this section, we interpret the results in two aspects: raw fault detection and crucial fault detection. We list vulnerable hardware structures where all crucial faults can be detected by the RAE ALU and the RAE MULTDIV. All these structures (such as registers and modules) have been explained in Chapter 3, hence we only introduce them briefly in this section.

4.5.1 Raw Fault Detection

Two residue bits are added to each of the 32 registers in the Register File, though register x0 is wired to zero. Residue arithmetic adds 64 residue bits, and associated combinational logic to the Ibex Core. There are 2072 bits in the Ibex Core enhanced by residue arithmetic.

As stated in subsection 4.2, we also explore faults in the extra residue bits. Table 4.2 shows the number of detectable bit locations by the ALU residue arithmetic (left part of the table) and the MULTIDIV residue arithmetic (right part of the table). Faults in the extra residue bits can also be detected.

TABLE 4.2: Detectable Bits by Residue Arithmetic

Property	Detectable Bits (ALU)	Property	Detectable Bits (MULTDIV)
SDC	1090	SDC	1132
Crash	1090	Crash	1132
Hang	1090	Hang	1132

Although three different groups of properties are used, in theory, the same residue arithmetic should be able to detect faults in the same bit locations because the detectable bit locations only depend on the implementation of residue arithmetic and the micro-architecture of the Ibex Core. In addition, formal verification is able to explore all the state space to find all the detectable bit locations. The principle applies to all the rows in Table 4.2. The RAE ALU can always detect faults in 1090 bit locations, and the RAE MULTDIV can always detect faults in 1132 bit locations, no matter the properties.

The RAE ALU and RAE MULTDIV can detect faults occurring in about half of the total bit locations in the Ibex Core. These detectable bits are more than bits in the Register File. We found that some bit locations in the IF Stage, ID Stage, LSU and MULTDIV (Figure 1.1) are also detectable. Table 4.3 shows the detectable registers in each module. A 'D' in the last two columns represents faults in the corresponding register that can be detected by the RAE ALU (`ra_alu`) and the RAE MULTDIV (`ra_mul`). In some registers such as `addr_last_q`, `mult_state_q` and `rf_reg_q`, faults in all bits can be detected. However, not all bits in the other listed registers are detectable. Most of these registers are in the second pipeline stage. Faults in both the control path (such as `controller`) and data path (such as `RegFile`) can be detected.

TABLE 4.3: Detectable Registers By Residue Arithmetic

Module	Register	ra_alu	ra_mul
controller	ctrl_fsm_cs		D
id_stage	imd_val_q		D
if_stage	instr_rdata_id_o	D	D
	instr_rdata_alu_id_o		D
load_store_unit	addr_last_q	D	
	ls_fsm_cs	D	D
MULTDIV	md_state_q		D
	mult_state_q		D
RegFile	rf_reg_q	D	D

4.5.2 Crucial Fault Detection

Table 4.4 shows the model checking results of exploring fault effects with the RAE ALU in the Ibex Core. The first column named 'Property' lists the proved properties which are also explored SEU effects. The second column named 'Failure' lists the

TABLE 4.4: Model Checking Results of Residue Arithmetic Enhanced ALU

Property	Failure	Proven
Instruction_is_done	993	97
Instruction	993	97
rs1_address	993	97
rs2_address	993	97
rd_address	993	97
rs1_read_data	993	97
rs2_read_data	993	97
rd_write_data	993	97
current_PC	993	97
next_PC	993	97
memory_address	993	97
memory_read_mask	993	97
memory_read_data	993	97
memory_write_mask	993	97
memory_write_data	993	97
Insn_access_fault	0	1090
Illegal_insn	0	1090
breakpoint	0	1090
load_access_fault	0	1090
store_access_fault	0	1090
ECall_MMode	0	1090
Hang	0	1090

number of detected crucial faults. The last column named 'Proven' lists the number of detected safe faults. There are three blocks in Table 4.4 representing the results of strobe, crash and hang properties respectively. As mentioned before, we use strobe properties because they can cover faults in more structures such as the faults in the Register File.

Note that the sum of each row in Table 4.4 is 1090, because only faults in the 1090 detectable bit locations were explored. Exploring faults in undetectable bit locations cannot raise the error indication signal `alu_ra_error`, hence all the crucial fault detection properties in subsection 4.3.2 can only report meaningless vacuous pass results.

TABLE 4.5: Vulnerable structures detected by Residue Arithmetic Enhanced ALU

Failure	Register	Module
SDC	ls_fsm_cs	load_store_unit
SDC	rf_reg_q	RegFile

Table 4.4 shows that the RAE ALU is good at detecting crucial faults that may lead to SDCs, but the RAE ALU cannot detect any faults that may lead to crashes or hangs. All the SDC crucial faults that can be detected by the RAE ALU are in the `ls_fsm_cs`

register in the `load_store_unit` module and the `rf_reg_q` register in the `RegFile` module, as shown in Table 4.5.

992 of the above detected crucial faults are in the Register File because there are 32 registers in the Registers File but the register x0 is wired to 0. All the faults, which may cause SDCs, encountered in the Register File can be detected by the RAE ALU. Another fault is in bit 862 in the `ls_fsm_cs` that stores the current state in the `ibex_load_store_unit` module. This module outputs the last transaction address, which determines $OP1$ to the ALU when operand forwarding is executed. Operand forwarding is a technique to avoid data hazards. For example, there are two consecutive instructions. The destination register in the first instruction is the source register in the second instruction. The pipeline must be stalled when executing the second instruction because the result of the first instruction has not been stored in the Register File at the current clock cycle. Operand forwarding allows the processor to bypass the stall by forwarding the result of the first instruction to the second instruction. A fault in the register `ls_fsm_cs` may stall the whole pipeline and produce the wrong last transaction address, hence wrong $OP1$ to the ALU when operand forwarding is executed. As a result, this fault is also a detected crucial fault by the RAE ALU.

TABLE 4.6: Model Checking Results of Residue Arithmetic Enhanced MULTDIV

Property	Failure	Undetermined	Proven
Instruction_is_done	1028	104	0
Instruction	1050	82	0
rs1_address	1026	106	0
rs2_address	1014	118	0
rd_address	1022	110	0
rs1_read_data	1023	109	0
rs2_read_data	1020	112	0
rd_write_data	1047	85	0
current_PC	1025	107	0
next_PC	1026	106	0
memory_address	1027	105	0
memory_read_mask	1007	125	0
memory_read_data	1001	131	0
memory_write_mask	984	148	0
memory_write_data	1026	106	0
Insn_access_fault	0	1119	13
Illegal_insn	12	1119	1
breakpoint	0	1132	0
load_access_fault	0	1120	12
store_access_fault	0	1116	16
ECall_MMode	0	1127	5
Hang	0	1129	3

Table 4.6 shows the model checking results of exploring fault effects with the RAE MULTIDV in the Ibex Core. The first column named 'Property' lists the proven assertions which are also explored SEU effects. The second column named 'Failure' lists the number of detected crucial faults. The third column named 'Undetermined' lists the number of undetermined results. The last column named 'Proven' lists the number of detected safe faults. There is a large number of undetermined results in Table 4.6 because we could not fully verify the multiplier and the divider in the MULTDIV when executing the RV32M instructions. The sum of each row in Table 4.6 is 1132, because only faults in the 1132 detectable bit locations were explored.

TABLE 4.7: Vulnerable structures detected by Residue Arithmetic Enhanced MULTDIV

Failure	Register	Module
SDC	imd_val_q	id_stage
SDC	instr_rdata_id_o	if_stage
SDC	instr_rdata_alu_id_o	if_stage
SDC	ls_fsm_cs	load_store_unit
SDC	md_state_q	MULTDIV
SDC	mult_state_q	MULTDIV
SDC	rf_reg_q	RegFile
Crash	instr_rdata_id_o	if_stage

On the other hand, the number of detected crucial faults that violate different SDC assertions in Table 4.6 are not all the same. Most of the detected crucial faults are in the Register File. Register `imd_val_q` in `id_stage` module contains the second most detected crucial fault. This register is an intermediate register for multiplications. It stores intermediate values during the MAC operations. A fault in this register can cause a wrong multiplication result. The RAE MULTDIV can also detect faults in some other registers, as shown in Table 4.7. For example, `instr_rdata_id_o` and `instr_rdata_alu_id_o` in `if_stage` module, `ls_fsm_cs` in `load_store_unit` module; `md_state_q` and `mult_state_q` in `MULTDIV` module. Both registers `instr_rdata_id_o` and `instr_rdata_alu_id_o` store the same instruction to `id_stage` module, the latter is a copy of the former to reduce fan-out. Both two registers are decoded to determine such as the ALU/MULTDIV control logic signals and associated operands. These detected crucial faults are not only in the data path but also in the control path. The register `ls_fsm_cs` has been explained before. `md_state_q` determines the state machine of division, `mult_state_q` is the state machine of multiplication. Faults in these two registers can lead to wrong division or multiplication results.

In Table 4.4, all the detected faults cannot cause any crashes or hangs. In other words, no detected crucial faults that may lead to crashes or hangs are found. In Table 4.6, only 50 faults that cannot cause any crashes or hangs, and 12 detected crucial faults that may lead to crashes are found; the rest are undetermined results. The detected crucial faults are in the register `instr_rdata_id_o` (the same register in the last paragraph). This proves that a fault in the control path can cause an arithmetic (multiplication or division) error and hence cause a crash.

By comparing the above results with the raw crucial faults (without any fault-tolerant technologies) in Section 3.8, we can find which crucial faults can be detected by residue arithmetic and which cannot be detected. For instance, in subsection 3.8.1.1 there are 1418 crucial faults that may corrupt the write data to DMEM, `memory_write_data`. The RAE ALU can detect 993 of them, and the RAE MULTDIV can detect 1026 of them.

After comparison, more than half of the crucial faults causing SDCs in Section 3.8 are identified in this subsection, hence detected by the residue arithmetic. However, faults in the most vulnerable structures discussed in Chapter 3, such as faults in the `FIFO` and the PC, cannot be detected by the implemented residue arithmetic.

On the contrary, only 12 crucial faults causing crashes are identified, and no crucial faults causing hangs are detected by the residue arithmetic. In addition, no faults in the extra residue bits can cause SDCs, crashes or hangs, because the extra residue bits are not consumed in the Ibex Core except by the residue arithmetic itself. In other words, faults in the extra residue bits cannot affect the functionality of the Ibex Core.

In summary, more than half of the bit locations in the Ibex Core are detectable by the implemented residue arithmetic, but not every fault in the detectable bit locations leads to failures. The implemented residue arithmetic cannot detect all crucial faults. The implemented residue arithmetic is good at detecting crucial faults that cause SDCs, but poor at detecting crucial faults that lead to crashes and hangs. The RAE MULTDIV have more detectable bit locations than the RAE ALU, and the RAE MULTDIV can detect more crucial faults than the RAE ALU.

4.6 Discussion and Conclusion

We proved strobe properties instead of performing SEC to identify faults that may cause SDCs in the presence of fault-tolerant technologies. The idea of strobe properties is similar to SEC. Both approaches insert 'strobes' into important signals. SEC compares strobes between the specification design (the golden design) and the implementation design (the actual design under test). Strobe properties compare strobes between the golden core and the faulty core. We did not choose SEC, because it is hard to control fault injection in the Cadence Jaspergold SEC App. In addition, the SEC App has limited support for proving user-developed assertions. As a result, we chose to perform model checking with strobe properties.

We evaluated the raw fault detection effectiveness and the crucial fault detection effectiveness of residue arithmetic. Residue arithmetic cannot detect faults in all registers in the Ibex Core. However, faults in more than half the bit locations in the Ibex Core can be detected by the residue arithmetic, which means the raw fault detection effectiveness is more than 50%. In addition, more than half the crucial SEUs that may lead to SDCs can be detected, which means the crucial fault detection effectiveness is also more than 50%. Residue arithmetic brings little hardware overhead. Residue arithmetic is good at detecting crucial SEUs that may cause SDCs, but poor at detecting SEUs that may lead to crashes and hangs. With extra fault correction technologies, the detected crucial SEUs can be protected. However, residue arithmetic cannot cover the most vulnerable structures such as `FIFO` and the PC. We also used the method to validate the framework in Section 3.7. The results are as expected, proving our method can be used to evaluate fault-tolerant technologies.

Applying our method to other RISC-V cores does not require developing properties and constraints again. The only modification is signal mapping. We also developed a script to automate the whole process. No extra formal knowledge and design efforts are needed. As a result, our method is easy-to-use.

One further application of this method is to use different technologies to protect different bits or registers according to the rank of reliability. For example, use the most expensive (and the most efficient) technology to protect the most vulnerable bits; cheaper technology to protect less vulnerable bits; and no protection for safe bits.

There are limitations to the current work. For instance, a long model checking run time and undetermined results, caused by state explosion and model checking multipliers and dividers. However, this work aims to prove that our method can be used to evaluate fault-tolerant technologies; conquering the challenge of formally verifying multipliers and dividers is out of the scope of this research.

Single Event Upsets have been a major reliability concern in embedded systems. ISO26262 suggests using simulation-based fault injection to improve safety and reliability [34]. Simulation-based methods suffer from fault coverage and long simulation time. To address the issue, we have proposed an analysis method that utilizes formal verification (mainly model checking) to evaluate reliability and safety. The proposed method can also evaluate the effectiveness of fault-tolerant technologies. The proposed method is exhaustive and time-saving compared to simulation-based fault injection. In addition, the proposed method is compatible with other RISC-V microprocessors. Users only need to map signals. There is no need to change the design or to develop formal properties and assumptions again. We demonstrate the method on a well-known fault-tolerant techniques: residue arithmetic. Residue arithmetic adds extra residue bits to the Register File. The results are as expected, which proves that the method can accurately explore and categorize all faults according to fault effects within a reasonable time. The experimental results show that residue arithmetic can detect faults injected at more than half of the bit locations in the Ibex Core. However, not all faults in the detectable bits lead to failures. Residue arithmetic can detect most of the crucial faults causing SDCs, a few crucial faults leading to crashes, and no faults leading to hangs. Residue arithmetic can detect crucial faults in both the control path and the data path, i.e., faults in both paths can lead to SDCs. However, residue arithmetic cannot detect any faults in the most vulnerable hardware structures such as the `FIFO` and the PC. As a result, although residue arithmetic is a cost-efficient fault-tolerant technology, it is not suitable to implement only residue arithmetic to enhance a safety-critical system. In conclusion, the proposed method can successfully evaluate fault-tolerant technologies, and would, in principle, be compatible with ISO26262.

Chapter 5

Double Event Upsets

In Chapter 3, the formal properties for revealing SEUs that lead to Silent Data Corruptions (SDCs), crashes and hangs were formulated. We demonstrated our formal method on a RISC-V Ibex Core. The whole state space was explored and all SEUs that lead to failures were identified. However, that formal method is only for SEUs. This Chapter aims to expand that formal method to Double Event Upsets (DEUs).

We call SEUs that cause errors and system failures, such as SDCs, crashes and hangs, *crucial* SEUs. We describe SEUs that have no effects as *safe* SEUs. Similarly, we define crucial DEUs and safe DEUs. The outcomes of model checking have been explained in Section 3.8. If the status of an assertion is 'Proven', faults in the corresponding bits cannot cause errors that violate the property. The faults in the bits are therefore deemed safe. If 'Failure', the faults in the corresponding bits may cause errors that violate the property and are crucial faults.

There are three hypotheses in Section 1.6. We aim to prove the third hypothesis (DEUs can aggravate SEUs) in this Chapter. We have the following objectives in this Chapter:

- To demonstrate that DEUs can aggravate SEUs.
- To identify all crucial DEUs that can cause SDCs, crashes, and hangs using model checking.
- To decrease prohibitively large model checking time to an acceptable range. One possible assumption is to exclude DEUs, that result from already-identified crucial SEUs, from the fault list. For instance, if one fault in the DEUs is already identified as a crucial SEU. Excluding this kind of DEU allows us to reduce the size of the search space significantly.
- To assess the reliability of a RISC-V Ibex Core in the presence of DEUs.

In this chapter, we make the following assumptions. The details will be given in later sections.

- We assume safe and crucial SEUs have been identified using the formal method proposed in Chapter 3 before utilizing the method proposed in this chapter to explore DEUs. The SEU results are used to prune the fault list of DEUs.
- We assume faults could occur in all bits equally. The details and reasons are in Section 5.2.
- We assume crucial SEUs and hence vulnerable bit locations will be protected. Thus, DEUs including crucial SEUs do not need further analysis and are removed from the fault list.
- Similarly, we assume crucial DEUs and corresponding vulnerable bits will be protected. Details and reasons are in Section 5.5.1.

5.1 Method Overview

The method in this chapter is extended from the method in Chapter 3. As a result, the basic idea is similar: using model checking to report all DEUs that may violate the developed property assertions.

The method to explore DEUs in this chapter can be divided into four steps, as shown in the following:

1. Implement a fault injection mechanism into the Ibex Core.
2. Formalize errors and system failures, such as SDCs, crashes and hangs as formal properties.
3. Develop various complexity control strategies to improve proof performance.
4. Perform model checking with a model checker to exhaustively identify and classify DEUs.

The first step is to modify the fault injection mechanism described in Section 3.2 and implement the modified version into the Ibex Core. The details of the modification will be given in Section 5.2.

The second step is to develop formal properties in SVAs. We have developed formal properties in Section 3.3, hence the details will not be repeated in this chapter. In Section 5.3, we will explain which properties are chosen to explore DEUs with reasons.

In the third step, the complexity control strategies from Section 3.4 are modified to explore DEUs. The details will be given in Section 5.4.

The final step is to perform model checking. Cadence JasperGold FPV is also chosen as the model checker. The experimental strategy and the formal tool settings are slightly different from those in Section 3.5. The details will be given in Section 5.5.

5.2 Step 1: Fault Injection

5.2.1 Fault Model

The last two chapters chose the SEU as the fault model, which has been introduced in Section 3.2. In this chapter, DEUs, which include two SEUs, are chosen as the fault model. Details of DEUs, such as three fault attributes (the location where a fault occurs, the time when the fault occurs, and the period for which the fault exists, Section 3.2), are similar to SEUs and hence not repeated.

We are working at the Register Transfer Level (RTL). The physical connection of bits is not considered; all bits are treated equally. In other words, we assume faults could occur in all bits equally. Another reason for ignoring the physical proximity is that we are working at the early design stages where the actual layout is unavailable. In addition, after identifying vulnerable bits using our method, designers can modify the physical proximity accordingly. Moreover, we mainly aim to prove the idea of our method. Ignoring the physical proximity helps to reduce complexity, which makes our problem solvable.

A DEU consists of two SEUs. We categorize DEUs into three types based on timing and location: two faults occurring in different bit locations and different times; different bit locations and the same time; the same location and different times.

5.2.2 Fault Injection Mechanism

Section 3.2 has introduced the fault injection mechanism to inject single faults. That fault injection mechanism is slightly modified so that it can inject DEUs. For example, previously there were two fault control signals (fault injection time and injection location) to control a single SEU. The fault control signals are doubled so that DEUs can be controlled via primary inputs. In addition, extra combinational logic is added so the two faults can be injected independently. The source code is shown at the bottom of Appendix D.

In subsection 3.2.3 we developed two SVA assumptions to specify an arbitrary SEU. Those SVA assumptions are modelled (by adding an extra pair of fault control signals) to specify an arbitrary DEU, as shown in the following:

```
assume_FI_time1_stable:
assume property ( @(posedge clk_i) $stable(FI_time1) );
assume_FI_time2_stable:
assume property ( @(posedge clk_i) $stable(FI_time2) );

assume_FI_index1_stable:
assume property ( @(posedge clk_i)
  $stable(FI_index1)&&(FI_index1<2008) );
assume_FI_index2_stable:
assume property ( @(posedge clk_i)
  $stable(FI_index2)&&(FI_index2<2008) );
```

We categorized DEUs into three groups in the last subsection. As a result, extra assumptions are developed to specify the equivalence of the two fault injection times and locations. The following assumptions determine whether the two faults are injected at the same time. This is the same as the fault locations, whose assumptions are not shown to save space.

```
`ifdef SameTime
  assume_FI_time_equal: //DEUs at the same clock cycle
  assume property ( @(posedge clk_i)
  (FI_time1 == FI_time2) );
`else
  assume_FI_time_different: //DEUs at different clock cycles
  assume property ( @(posedge clk_i)
  (FI_time1 != FI_time2) );
`endif
```

5.3 Step 2: Formal Properties

In Section 3.3 we developed three groups of properties that cover SDCs, crashes and hangs. Those properties are used without any modification in this chapter to explore DEUs. There are two different groups of properties to explore SDCs: architectural properties and strobe properties. Architectural properties are used to explore DEUs. The first reason for choosing architectural properties is that strobe properties are not suitable for exploring DEUs, because strobe properties significantly increase the design size, leading to greater runtime overhead.

The second reason is that verifying at the architectural level allows abstracting and neglecting unnecessary structures when proving an assertion. For example, neglecting structures outside the COI of the assertion, and abstracting a counter. Abstracting counters in formal verification is a useful technology to reduce the complexity and large state space caused by counters. The formal tool can automatically perform counter abstraction by either replacing the initial/reset value with an arbitrary value so that the counter can start counting from an arbitrary value or only considering critical counter values such as trigger values that may have an influence on the design.

The third reason is that although proving strobe properties reports a higher fault coverage than proving architectural properties, all the SEUs in the structures (such as the `RegFile` module) that cannot be covered by architectural properties are crucial SEUs. We assume crucial SEUs and hence vulnerable bits will be protected. The structures that cannot be covered by architectural properties are vulnerable and hence do not need further analysis. DEUs including crucial SEUs in the vulnerable structures do not need further analysis and are removed from the total fault space. As a result, architectural properties are enough to explore the DEUs that include only safe SEUs.

5.4 Step 3: Complexity Control Strategies

We developed four complexity control strategies in Section 3.4. Most of the strategies are used without modification to analyse DEUs, except the second and the fourth strategies. To save space, we only introduce the modifications in this section.

5.4.1 Input Constraints

In Section 3.4.2, we developed assumptions of the input instructions to ensure that: 1) only bit patterns corresponding to valid RV32IMC instructions are allowed; and 2) only aligned RV32IMC instructions are allowed. However, we did not fully prove architectural properties that formulate RV32M instructions, because fully verifying multipliers and dividers using formal verification is a challenge and out of the scope of this research. We did not find all safe SEUs with respect to RV32M instructions, hence we cannot precisely prune the DEU list of RV32M instructions. In addition, model checking the architectural RV32M properties in the presence of DEUs is even more complex than SEUs. As a result, we exclude RV32M from the assumptions when analysing DEUs. For example, if the instruction address from the Ibex Core is aligned by 4, then input a valid RV32I instruction to the Ibex Core; otherwise, input a valid RV32C instruction to the Ibex Core.

5.4.2 Handle Undetermined Results

In subsection 3.4.4 we introduced how to use case splitting to analyse undetermined results. We divided the whole fault space into several sub fault lists at the module level, register level and even bit level (Figure 3.8) to disprove or fully prove undetermined results.

Given a design with n modules, the whole SEU space can be divided into n sub SEU lists, hence only n extra model checking runs are required to explore undetermined SEUs. However, there are two faults in a DEU, hence producing n^2 sub DEU lists and n^2 extra model checking runs to explore undetermined DEUs. We developed a script that contains a nested loop to explore each combination in the n^2 sub DEU lists.

5.5 Step 4: Model Checking

The experimental strategy and the formal tool settings have been explained in Section 3.5. We only introduce the difference in this section.

5.5.1 Experimental Strategy

The strategy to explore DEUs can be briefly divided into the following steps:

> For each assertion:
>
> a) Start with an arbitrary DEU;
>
> b) Perform model checking to identify a crucial DEU that violates the assertion;
>
> c) Record the crucial DEU and remove it from the fault space;
>
> d) Go back to b) with the updated fault space. Repeat until no more crucial DEU can be found and the assertion is fully proven.

There are 2008 bits in the Ibex Core. In theory, there are about four million possible fault location combinations of DEUs in the Ibex Core. However, combinations including crucial SEUs are excluded. The SEU experimental strategy in subsection 3.5.1 must be performed first to identify crucial SEUs before performing the DEU experimental strategy. The identified SEUs have been given in Section 3.8. We assume it is unnecessary to further analyse DEUs that include crucial SEUs.

We have shown how to identify crucial SEUs and vulnerable bit locations using our formal method in Chapter 3. If we use, for example, TMR to protect those vulnerable bits further faults in those bits have no effect. Thus, DEUs including faults in those bit locations (DEUs including crucial SEUs) do not need further analysis because these bits have already been identified as vulnerable bits. Without this assumption, DEUs including crucial SUEs will be explored, which is essentially exploring the crucial SEUs again. As a result, it is not necessary to explore DEUs including crucial SEUs. Only DEUs excluding crucial SEUs are explored, to examine whether DEUs can aggravate SEUs.

Since the crucial SEUs of different properties are not all the same, the initial DEU list of each assertion to explore DEUs is different. As a result, in step a), an arbitrary DEU is constrained to a smaller fault space where only safe SEUs are included. This is achieved by assumptions. The following is an example. When verifying the `RV32I_LUI` property in the presence of DEUs, all crucial SEUs are excluded in step a) of the DEU experimental strategy.

```
`ifdef RV32I_LUI
  assume property (
  !(FI_index1 inside {/*all crucial SEU bits to RV32I_LUI*/}));
  assume property (
  !(FI_index2 inside {/*all crucial SEU bits to RV32I_LUI*/}));
`endif
```

In step c) the identified DEUs from step b) are added to constraints (as SVA assumptions) in order to exclude them from further model checking runs. For example, after finding one crucial DEU violating an assertion, this identified DEU is excluded from the fault list the next time proving the same assertion. This step aims to avoid repeatedly identifying the same DEUs. It also reduces the fault list and forces the formal tool to explore the other DEUs. The following is an example of assumptions about an assertion after identifying two crucial DEUs.

```
`ifdef insn_access
  assume_FI_index1_stable:
  assume property (
  $stable(FI_index1) && (FI_index1<2008)
  && !(FI_index1 inside {509,510}) );

  assume_FI_index2_stable:
  assume property (
  $stable(FI_index2) && (FI_index2<2008)
  && !(FI_index2 inside {509,510}) );
`endif
```

We use the range from 0 to 2007 to represent the register bits in the Ibex Core. The bit index is shown in Appendix E. In the above example, the already identified crucial DEUs are in bit (509,509) and bit (510,510). These two bits are in the register `valid_q`, which determines the validity in the `FIFO` module. `FI_index1` and `FI_index2` controls fault injection bit locations of DEUs. `!(FI_index1 inside {509,510}))` makes sure the already identified crucial DEUs can no longer be explored in further model checking runs.

It might be argued that the above assumptions also exclude DEUs in bits (509,510) and bits (510,509) from the fault list, hence reducing the result accuracy. However, protection targets are bits instead of DEUs. We argue that since the bits have already been identified as vulnerable bits and need protection to mitigate DEUs, it is unnecessary to further explore DEUs that include one of the vulnerable bits. Similar to protecting SEUs, we assume that after identifying one crucial DEU, the corresponding vulnerable bit locations will be protected. For example, bits 509 and 510 are enhanced by TMR. Exploring a DEU, that includes one enhanced bit and one unprotected bit, is essentially exploring an SEU in the unprotected bit. As a result, it is unnecessary to further explore DEUs including any of the vulnerable bits. This reduction also contributes to saving model checking time and computation efforts.

5.5.2 Configurations of Cadence JasperGold

The tool settings were the same as the previous settings in subsection 3.5.2, except the proof time was changed to five hours, because we found some cases took up to five hours to be fully proved.

The same engine modes in subsection 3.5.2 were used to perform model checking in the presence of DEUs. The only difference is that case splitting and the abstraction engines AD and AM were used more frequently when exploring DEUs. For example, most of the properties were fully proved by AM in the presence of DEUs. In addition, it was faster to use AD to find counterexamples than Hts, hence AD was mainly used to identify crucial DEUs. Similarly to Chapter 3, it was the easiest to prove Integer Computational Instructions, it was harder to prove Control Transfer Instructions, and it was the most difficult to prove Load and Store Instructions. In general, formally analysing DEUs is harder than formally analysing SEUs.

5.6 Results

DEUs can occur in one pair of bits but at different times. In other words, there may exist a very large number of DEUs from the same bit pairs. We argue that in each bit pair, one DEU is enough to prove the bits are vulnerable to DEUs. Similar to SEUs, we assume that after identifying vulnerable bits, fault-tolerant technologies will be implemented in these identified vulnerable bits, hence all DEUs in these bits can be protected. Thus, it is not necessary to further explore these DEUs in the same bit pair.

During the experiments, we found that an assertion may fail if one of the double faults is a crucial SEU (regardless of the other fault). Exploring such DEUs is in fact exploring crucial SEUs, which is unnecessary. We are interested in errors/failures caused by DEUs rather than SEUs. As stated before, we assume that bits, where crucial SEUs may occur, will be protected. As a result, when proving an assertion, DEUs, that include corresponding crucial SEUs are excluded from the fault list.

In this section, only necessary SEU results, such as crucial SEUs, are presented. Detailed SEU results and how they are identified have been shown in Section 3.8. We identified 11696 crucial SEUs causing SDCs, 159 crucial SEUs causing crashes, and 10 crucial SEUs causing hangs. These crucial SEUs are used for DEU reduction. As noted above, we exclude any DEUs that include these crucial SEUs. For example, when proving the hang property, DEUs including the 10 crucial SEUs causing hangs are excluded from the fault list. As noted above, each DEU assertion has a unique DEU fault list. We use SVA assumptions to specify the DEUs lists. Only DEUs that include safe SEUs will be explored.

As stated in Section 5.2, we categorized DEUs into three groups: two faults occurring in different bit locations and at different times; different bit locations and at the same time; the same location and at different times. We used our method to explore each group. All the results are either proven or disproved. Though there are two pipeline stages in the Ibex Core, after injecting a DEU, it can take more than two clock cycles to report a counterexample because the DEU can stall or flush the pipeline. On average, for 99% properties, it only took three minutes to report a counterexample (step b in subsection 5.5.1). The remaining 1% were disproved with case splitting. It took up to an hour to disprove each case. All the crucial DEUs were falsified by Hts and AD. It took up to five hours to fully prove a property for a given DEU list (step d in subsection 5.5.1). Most of the safe DEUs were proved by AM and the others were proved by Hps. Case splitting (in subsection 5.4.2) was used to handle all properties. With the help of case splitting, all the properties can be fully proved. It took on average one hour to fully prove each case in the RV32IC Integer Computational Instruction properties. It took one and a half hours to fully prove 80% of cases in the RV32IC Control Transfer and Load and Store Instruction properties. However, it took three hours to fully prove about 17% of cases and up to five hours to fully prove the 3% of cases, which is similar to crash and hang properties.

We interpret the results at two levels: hardware-level and instruction-level. Hardware-level identifies structures (such as bits, registers and modules) vulnerable to DEUs; hardware protection technologies, such as TMR, can be implemented to protect these vulnerable bits. Instruction-level identifies instructions vulnerable to DEUs; software protection technologies, such as redundant instructions, can be implemented to protect these vulnerable instructions. Apart from software, we found enhancing specific bits/registers/modules can also improve the reliability of instructions.

5.6.1 Hardware-level

TABLE 5.1: Overview results of crucial DEUs

Property	DLDT	DLST	SLDT	Property	DLDT	DLST	SLDT
RV32I_LUI	66	76	0	RV32I_SW	4	8	1
RV32I_ADD	10	2	0	RV32I_SB	4	4	1
RV32I_ADDI	2	2	0	RV32I_SH	4	4	1
RV32I_XOR	10	2	0	RVC_LUI	26	12	0
RV32I_XORI	2	2	0	RVC_ADD	118	58	0
RV32I_OR	40	12	0	RVC_ADDI	25	20	0
RV32I_ORI	28	10	0	RVC_XOR	22	20	1
RV32I_AND	40	12	0	RVC_OR	20	18	1
RV32I_ANDI	28	10	0	RVC_AND	101	34	0
RV32I_SLL	10	2	0	RVC_ANDI	55	28	0
RV32I_SLLI	2	2	0	RVC_SLLI	89	38	0
RV32I_SRL	10	2	0	RVC_SRLI	115	48	0
RV32I_SRLI	2	2	0	RVC_SUB	108	40	0
RV32I_SUB	150	24	0	RVC_LI	2	14	0
RV32I_SRA	145	24	0	RVC_MV	83	48	0
RV32I_SRAI	38	6	0	RVC_ADDI4SPN	74	30	0
RV32I_AUIPC	61	71	0	RVC_ADDI16SP	47	26	0
RV32I_SLT	38	12	0	RVC_JAL	48	26	0
RV32I_SLTI	26	10	0	RVC_JALR	534	72	0
RV32I_SLTIU	26	10	0	RVC_BEQZ	25	20	0
RV32I_SLTU	38	12	0	RVC_BNEZ	67	60	0
RV32I_JAL	2	2	0	RVC_JR	25	20	0
RV32I_JALR	2	2	0	RVC_J	43	26	0
RV32I_BEQ	29	6	0	RVC_LW	4	8	0
RV32I_BNE	2	2	0	RVC_LWSP	6	10	0
RV32I_BLT	2	2	0	RVC_SW	11	12	0
RV32I_BGE	2	2	0	RVC_SWSP	13	14	0
RV32I_BLTU	2	2	0	Insn_access_fault	0	0	0
RV32I_BGEU	2	2	0	Illegal_insn	22	20	2
RV32I_LW	4	8	0	breakpoint	78	66	2
RV32I_LB	5	8	0	load_access_fault	27	4	0
RV32I_LBU	5	8	0	store_access_fault	27	4	0
RV32I_LH	5	16	0	Ecall_MMode	1361	7858	6
RV32I_LHU	4	6	0	Hang_WFI	1361	85	27

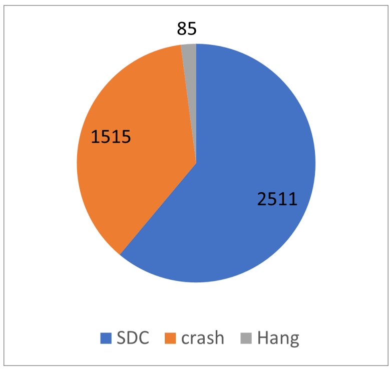


FIGURE 5.1: Overview: crucial DEUs in different bits at different times

Table 5.1 lists the results of proving all properties in the presence of DEUs. To save space, only failures are listed. There are no undetermined results. The first and the fifth columns list the property names. The remaining columns list the number of crucial DEUs in different groups. 'DLDT' represents DEUs in different bits at different times, 'DLST' represent DEUs in different bits at the same time, and 'SLDT' represent DEUs in the same bit at different times. We will explain the results in detail (including the bits, registers and modules where the crucial DEUs occur) in the following subsections.

5.6.1.1 Different Bits, Different Times

The first experiment was to explore DEUs occurring in different bits at different times. The fault injection order was also considered. Two faults f_1, f_2 are injected in bits b_1, b_2 at clock cycles t_1, t_2, respectively. We assume that f_1 is always injected before f_2, $t_1 < t_2$.

We treat two DEUs occurring in the same bit pairs but in a different order as different DEUs, irrespective of fault injection times. For example, a DEU in b_1, b_2 and a DEU in b_2, b_1 are two different DEUs.

We treat DEUs occurring in the same bit pair and in the same order as the same DEU, no matter the fault injection times. For instance, the DEU injected in bit_1, bit_2 at clock cycles 6 and 7, and the DEU injected in bit_1, bit_2 at clock cycle 8 and 9, are the same DEU.

In this category, we found 4111 crucial DEUs. Figure 5.1 shows the effects of the identified crucial DEUs. 2511 DEUs cause SDCs, 1515 DEUs cause crashes, and 85 DEUs cause hangs. Figure 5.2 shows the number of the crucial DEUs that cause crashes. Among the DEUs causing crashes, 1361 DEUs cause Ecall crashes, 78 cause breakpoint crashes, 22 cause illegal instructions, 27 cause load- and store-access faults.

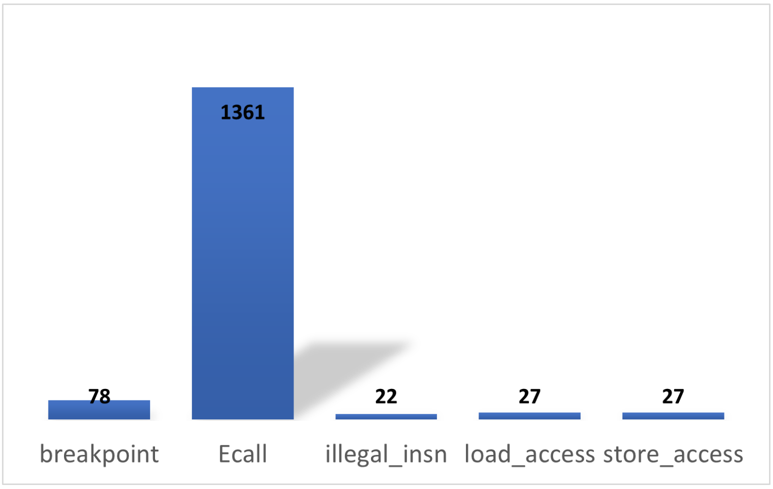


FIGURE 5.2: DEUs in different bits at different times can cause crashes

We found that DEUs in some bit pairs can lead to multiple assertion failures. In other words, because each assertion represents an error, these DEUs can lead to multiple errors, and these bit pairs are vulnerable to multiple errors. Table 5.2 lists the DEUs that cause multiple assertion failures. To save space, only DEUs that cause more than 16 assertion failures are listed here. The second column '# Fail' is the number of assertion failures the DEUs listed in the first column can cause. The bit numbers are in the first column. We use numbers to label all the bits in the Ibex Core, as shown in Appendix E. The bit ordering determines the injection ordering. For example, (867,2007) means the first fault is injected to bit 867 and the second fault is injected to bit 2007. So (867,2007) and (2007,867) represent two different DEUs. Note that the captured errors appear several clock cycles after the second fault is injected. Considering that only safe SEUs are included in DEUs, these errors are indeed caused by DEUs instead of the first safe SEU.

TABLE 5.2: The most frequent crucial DEUs in different bits at different times

DEUs	**# Fail**
(867, 2007)	45
(3, 2)(2007, 867)	32
(2007, 866)(1010, 866)(1009, 866)	20
(2,1009)(2,1010)(3,1009)(3,1010)(866,8)(866,9)	18
(866,1009)(866,1010)(866,2007)(1009,2)(1009,3)	18
(1010,2)(1010,3)(2007,2)(2007,3)	18
(512,750)(513,750)(528,750)(529,750)(544,750)(545,750)	16
(560,750)(561,750)(576,750)(577,750)(592,750)(593,750)	16
(750,1009)(750,1010)(750,2007)(1009,750)(1010,750)	16
(863,864)(2007,750)	16

It can be seen that the DEU at the bit pair (867,2007) (two safe SEUs injected at bit 867 then at bit 2007) causes the most failures. Bit 867 is a 1-bit register `lsu_err_q` in the `load_store_unit` module. This register records the error status of this module. Bit

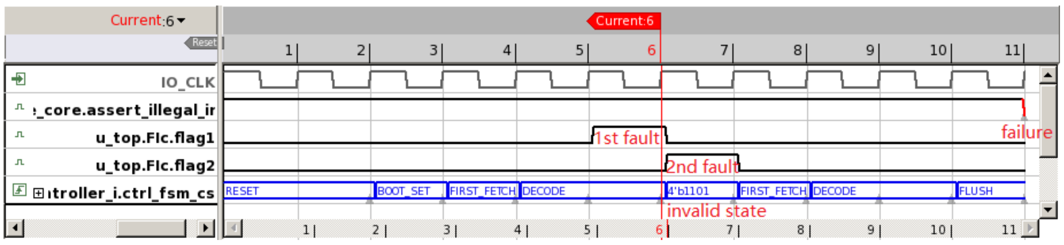


FIGURE 5.3: DEUs in bits (3,2) causing a crash

2007 is a 1-bit register `core_busy_q` in the top module of the Ibex Core. This register disables clock gating if it is logic high. Disabling clock gating leads to a pause in the Ibex Core.

The next most serious DEUs are encountered in bits (3,2) and bits (2007, 867). A DEU in bits (2007, 867) causes fewer failures than a DEU in bits (867, 2007). In other words, the ordering of fault injection influences whether a DEU can lead to errors or not.

Bits 3 and 2 are the two most significant bits in the register `ctrl_fsm_cs` in the `controller` module. This register stores the current state of the module. One of the errors caused by DEUs in bits (3,2) is a crash due to an illegal instruction. An illustration of this example is shown in Figure 5.3. After the `first_fetch` state, the state machine in the `controller` module should remain in the `DECODE` state. However, two consecutive faults in bits (3,2) force the state machine to enter an invalid state, `4'b1101` at the sixth clock cycle in Figure 5.3. The Ibex Core treats the next instruction as the first instruction (which it is not). The pipeline is refreshed, and data is lost. As a result, after several clock cycles, an exception is encountered, causing the assertion failure.

TABLE 5.3: Vulnerable structures containing the DEUs in Table 5.2

Module	**Registers**
`top`	`core_busy_q`
`controller`	`ctrl_fsm_cs`
`if_stage`	`instr_rdata_c_id_o`
`prefetch_buffer`	`rdata_outstanding_q`
`FIFO`	`rdata_q`
`load_store_unit`	`ls_fsm_cs,lsu_err_q,pmp_err_q`

Based on the identified vulnerable bit pairs, the vulnerable structures can be found. Table 5.3 lists the registers and modules that include the crucial DEUs in Table 5.2. Bit 750 is the LSB of the register `instr_rdata_c_id_o`, which stores the compressed instruction to the second pipeline stage. Bits 1009 and 1011 consist `rdata_outstanding_q`, which (together with other signals) affects the `FIFO`. Register `rdata_q` contains the bits [512:607], which stores the values in the `FIFO`. The previous work proves not all bits in the `FIFO` are crucial SEUs to failures, hence some bits in the

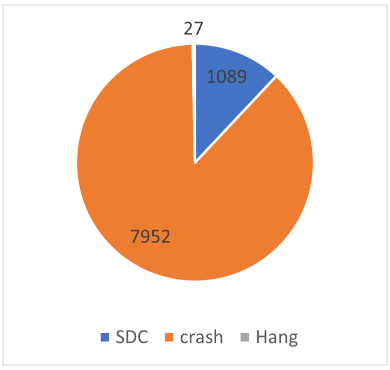


FIGURE 5.4: Overview: crucial DEUs in different bits at the same time

rdata_q are in the DEU lists. Bits 863,864 are in the ls_fsm_cs which is the state machine of the load_store_unit module.

Among the above examples, the combination lsu_err_q with core_busy_q, and the register ctrl_fsm_cs are the most vulnerable combinations to SDCs in the presence of DEUs. In addition, modules prefetch_buffer (mainly registers branch_discard_q and valid_req_q) and FIFO (mainly register err_q) are the most vulnerable structures to crashes and hangs in the presence of DEUs. As a result, in order to mitigate SDCs, crashes and hangs caused by DEUs in this category, it is important to enhance these structures with fault-tolerant technologies or place them away to reduce the possibility of being hit by the same energetic particle.

5.6.1.2 Different Bits, Same Time

The second experiment explored DEUs occurring in different bits at the same time. For example, $b_1! = b_2, t_1 == t_2$. After finding a DEU in one pair of bits, we assume these bits would be protected. Thus this pair is not further explored. One counterexample is enough to prove these bits are vulnerable to DEUs and need protection.

Figure 5.4 gives an overview of identified crucial DEUs for this case. 9068 crucial DEUs are found. 1089 DEUs cause SDCs, 7952 DEUs cause crashes, and 27 DEUs cause hangs. Unlike the last category, the most crucial DEUs in this category cause crashes.

Of the DEUs causing crashes, 7858 DEUs cause Ecall crashes, 66 cause breakpoint crashes, 20 cause illegal instructions, and 4 cause load- and store-access faults, as shown in Figure 5.5. Ecall is the most frequent crash caused by the DEUs in this category. Among the DEUs causing Ecall crashes, bits 509 and 511, which are inside the register valid_q in the FIFO module, are the most vulnerable bits. This register determines whether an instruction stored in the instruction FIFO (which stores fetched instructions from IMEM) is valid or not. There are 3978 crucial DEUs including faults

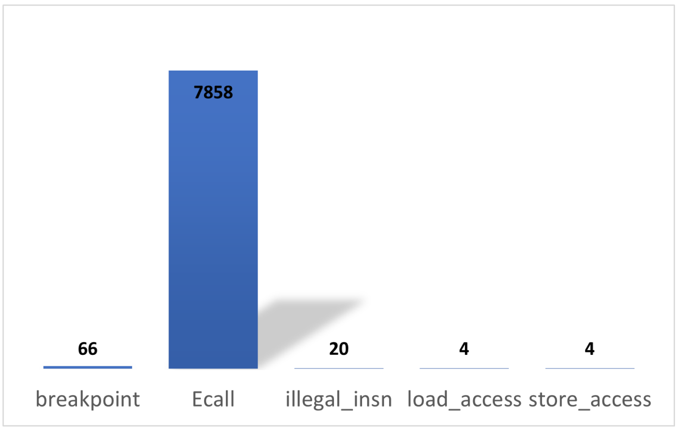


FIGURE 5.5: DEUs in different bits at the same time can cause crashes

encountered in bit 509, and 3170 crucial DEUs including faults encountered in bit 511, that can cause Ecall crashes. Bit 688 is the most common bit in crucial DEUs causing breakpoint crashes. 36 crucial DEUs including faults in bit 688 may cause breakpoint crashes. This bit is in the register `instr_rdata_id_o` in the `if_stage` module. This register stores the instructions to the `id_stage` module. Bits in this register are also the majority bits of crucial DEUs causing hangs. There are 12 DEUs including faults in bit 1011, and 10 DEUs including faults in bit 1012, that can cause illegal instructions. Bits 1011 and 1012 are in the register `branch_discard_q` in the `prefetch_buffer` module. This register determines whether the module pushes a new entry to the instruction `FIFO`. DEUs causing crashes are mainly in the instruction fetch and decode stages.

Table 5.4 lists the crucial DEUs in this group that can cause more than 12 multiple assertion failures. The others are not listed to save space. The bit numbers are listed in the first column. The second column '# Fail' shows the number of failures the DEUs listed in the first column can cause. Most of the bits are also in Table 5.2, except bit 681 in register `id_fsm_q` (the state machine) in `id_stage` module, hence they are not repeated.

TABLE 5.4: The most frequent crucial DEUs in different bits at the same time

DEUs	**# Fail**
(867, 2007)	45
(2,1009)(2,1010)(3,1009)(3,1010)(3,2007)	18
(866,1009)(866,1010)(866,2007)	18
(750,1009)(750,1010)(750,2007)	16
(751,766)	13
(863,681)(863,2007)	12

It can be seen that DEUs in bits (867,2007) in registers `lsu_err_q` and `core_busy_q` cause the most assertion failures, which is the same as in the previous category. Bits 867 and 2007 have been explained in the last subsection. The next most frequent DEUs occur in registers`ctrl_fsm_cs`, `rdata_outstanding_q` and `instr_rdata_c_id_o`.

These registers also affect the functionality of the `FIFO`. DEUs in the above bit/register combinations produce the most SDCs, hence enchaining them or putting them away should contribute to lower SDC rates. Modules `prefetch_buffer` (mainly registers `branch_discard_q` and `valid_req_q`) and `FIFO` (mainly register `err_q`) are the most vulnerable structures to crashes and hangs. Placing them away or enhancing them can reduce crash and hang rates.

We found that certain bits appear in many of the DEUs (including DEUs not shown in Table 5.4). For instance, there are 3978 crucial DEUs that occur in bit 509 and there are 3170 crucial DEUs that occur in bit 511. These bits are in the `valid_q` in the `FIFO` and determine the validness of values in the `FIFO`. These bits are the most vulnerable bits to the DEUs in this category. Table 5.5 lists the bits that are vulnerable to more than 100 DEUs. The second column lists the bit numbers. The third column lists the number of the DEUs that include the bits in the second column. Bit 2007 is in the top module and enables clock gating. There are 289 crucial DEUs that may occur in this bit. All of the remaining vulnerable are in the IF stage (first column). IF stage is the most vulnerable stage to DEUs in different bits at the same time.

TABLE 5.5: The most vulnerable structures containing a DEU in different bits at the same time

Stage	**Register**	**Bit**	**DEUs**
IF	`valid_q`	509	3978
IF	`valid_q`	511	3170
TOP	`core_busy_q`	2007	289
IF	`rdata_outstanding_q`	1010	264
IF	`rdata_outstanding_q`	1009	228
IF	`instr_rdata_c_id_o`	750	152
IF	`instr_rdata_id_o`	688	126
IF	`pmp_err_q`	866	166
IF	`ctrl_fsm_cs`	3	108
IF	`instr_rdata_id_o`	690	101

5.6.1.3 Same Bit, Different Times

The third experiment explored DEUs occurring in the same bit at different times. For example, $b_1 == b_2, t_1! = t_2$. For each bit, at most one crucial DEU is explored. After finding a DEU in a bit, we assume this bit would be protected. Thus this bit is not further explored. One counterexample is enough to prove a bit is vulnerable to DEUs.

There are fewer crucial DEUs in this category compared to the others. Only 18 crucial DEUs that occurred in the same bit at different times were found. 5 DEUs cause SDCs, 10 DEUs cause crashes, and 3 DEUs cause hangs. Most of the crucial DEUs cause crashes. Table 5.6 lists all crucial DEUs in this category. Since DEUs are injected at the same bit, only one bit number is listed. The first column lists three types of crashes and a hang, and the third column lists types of SDCs. For example, a_RV32I_S* represents crucial DEUs that may violate assertions that monitor the behaviours of RV32I_SB, RV32I_SH and RV32I_SW.

TABLE 5.6: Bits vulnerable to DEUs in the same bit location

Error	DEUs	Error	DEUs
Breakpoint	1011,1012	a_RV32I_S*	866
Ecall	509,511,544,545,576,577	a_RVC_XOR	741
Illegal_insn	1011,1012	a_RVC_OR	741
Hang_WFI	511,1011,1012		

The crucial DEUs in this category can only cause certain errors. As shown in Table 5.6, the DEUs can only cause one type of hangs, three types of crashes, and five types of SDCs. The crucial DEUs are in certain bit locations. Bit 866 (in register pmp_err_q in the load_store_unit), 511 (in register valid_q in the FIFO), 1011 & 1012 (in register branch_discard_q in the prefetch_buffer), and 0 (in register ctrl_fsm_cs in the ibex_controller) are the most vulnerable bits. All these bit locations have been explained in the last two subsections, hence not repeated.

5.6.2 Instruction-level

The results from architectural properties, which model correct instruction behaviour, show that some assertions identify more crucial DEUs than others. In other words, some instructions are more vulnerable to DEUs than others. Implementing software protection technologies to protect these instructions should greatly reduce failures due to DEUs.

There are two jump-related branch instructions in 16-bit compressed form: RVC_JR (jump) and RVC_JALR (jump and link). RVC_JR performs an unconditional control transfer. The target address is determined by the offset in the instruction and the current PC address. RVC_JALR also performs the same unconditional control transfer and writes the address of the instruction following the jump (PC+2) to the link register.

We found that DEUs affecting the compressed jump and link instruction RVC_JALR cause the most SDCs. For example, 606 DEUs may cause RVC_JALR fails. It is necessary to apply software protection to this instruction.

We also found that instructions that perform similar functions are vulnerable to the same crucial DEUs. Most of the vulnerable bits have been introduced in Section5.6.1, hence this subsection represents DEUs in the form of registers for a better view. The DEUs are not only at bits within the same register but also at bits across different registers. Table 5.7 lists the registers (in the last two columns) where injected DEUs can corrupt the instructions (in the first column). The DEUs are in different bits at either different times ('Dt') or the same time ('St').

TABLE 5.7: DEUs in certain registers can corrupt similar instructions

Instructions	Registers(Dt)	Registers(St)
RV32I Control and Transfer	lsu_err_q core_busy_q	lsu_err_q core_busy_q
RV32I Load and Store	ctrl_fsm_cs ls_fsm_cs	ctrl_fsm_cs id_fsm_q ls_fsm_cs
RVC Control and Transfer	ctrl_fsm_cs load_err_q store_err_q pmp_err_q lsu_err_q rdata_outstanding_q core_busy_q	ctrl_fsm_cs pmp_err_q lsu_err_q rdata_outstanding_q core_busy_q
RVC Load and Store	ctrl_fsm_cs pmp_err_q lsu_err_q	ctrl_fsm_cs id_fsm_q pmp_err_q instr_rdata_c_id_o instr_is_compressed_id_o addr_last_q ls_fsm_cs core_busy_q

The RV32I Control Transfer instructions `RV32I_BEQ`, `RV32I_BNE`, `RV32I_BLT`, `RV32I_BLTU`, `RV32I_BGE`, `RV32I_BGEU`, `RV32I_JAL` and `RV32I_JALR` are all vulnerable to DEUs (at either the different times or the same time) in registers `lsu_err_q` in the `load_store_unit` module and `core_busy_q` in the top module. The RV32I Load and Store instructions `RV32I_LW`, `RV32I_LH`, `RV32I_LHU`, `RV32I_LB`, `RV32I_LBU`, `RV32I_SW`, `RV32I_SH`, `RV32I_SB` are all vulnerable to DEUs at different times in registers `ls_fsm_cs` in the `load_store_unit` and `ctrl_fsm_cs` in the `controller`; and DEUs at the same time in registers `ls_fsm_cs` in the `load_store_unit`, `id_fsm_q` in the `id_stage` and `ctrl_fsm_cs` in the `controller`.

The RVC Control Transfer instructions `RVC_BEQZ`, `RVC_BNEZ`, `RVC_J`, `RVC_JAL`, `RVC_JALR` and `RVC_JR` are all vulnerable DEUs at different times in registers `ctrl_fsm_cs, load_err_q, store_err_q` in the `controller`, `pmp_err_q, lsu_err_q` in the `load_store_unit`, `rdata_outstanding_q` in the `prefetch_buffer` and `core_busy_q` in the top; and DEUs at the same time in registers `ctrl_fsm_cs` in the `controller`, `pmp_err_q, lsu_err_q` in the `load_store_unit`, `rdata_outstanding_q` in the `prefetch_buffer` and `core_busy_q` in the top.

The RVC Load and Store instruction RVC_LW, RVC_LWSP, RVC_SW and RVC_SWSP are all vulnerable to DEUs at different times in registers ctrl_fsm_cs in the controller, pmp_err_q, lsu_err_q in the load_store_unit; and DEUs at the same time in registers instr_rdata_c_id_o, instr_is_compressed_id_o in the if_stage, ctrl_fsm_cs in the controller, id_fsm_q in the id_stage, addr_last_q, ls_fsm_cs, pmp_err_q in the load_store_unit, and core_busy_q in the top.

In addition, some structures are more vulnerable to DEUs when executing compressed instructions rather than base instructions. For example, DEUs (in different bits at different times) in bits (3,2) (in the ctrl_fsm_cs) can cause 22 SDCs due to wrong results from compressed instructions and 8 SDCs due to wrong results from base instructions. One reason is that DEUs force the state machine in the controller module to restart from the beginning if it reads source register values incorrectly. If the instructions executed in the design are known, extra hardware protection technologies should be applied to these parts.

One application of the above finding is that designers can enhance certain instruction sets (mitigate SDCs caused by malfunctions of certain instruction sets) by focusing on specific hardware. For example, the structures in Table 5.7 should be enhanced to ensure the functionalities of RV32I and RVC Control Transfer and Load and Store instructions in the present DEUs, and hence mitigate SDCs caused by the malfunctions of these instructions.

Some instructions in base form (RV32I) and compressed form (RVC) perform similar functions, such as RV32I_ADD and RVC_ADD. We compared the DEUs in different bit locations at different times that corrupt RV32I instructions and the DEUs that corrupt RVC instructions. We observed that logical instructions and jump instructions in base form tend to be less vulnerable to DEUs than similar instructions in compressed form. Figure 5.6 shows the comparison. The blue bar is the number of DEUs that corrupt the correct behaviours of RV32I instructions, the orange bar is the number of DEUs that corrupt the correct behaviours of RVC instructions. The orange bars are longer than the blue bars in the logical instructions (ADD, ADDI, AND, ADNI, XOR) and jump instructions (JAL, JALR). There is a similar trend for DEUs in different bits at the same time, as shown in Figure 5.7.

On the other hand, load and store instructions in the base form and similar instructions in the compressed form, such as RV32I_LW & RVC_LW and RV32I_SW & RVC_SW, are vulnerable to the same DEUs. In real hardware, if software characteristic is known, then this finding would help to determine appropriate software protection technologies. For example, if the hardware executes both base and compressed form instructions and performs a lot of logical and branch instructions, then it would be necessary to apply software protection to compressed logical and compressed branch

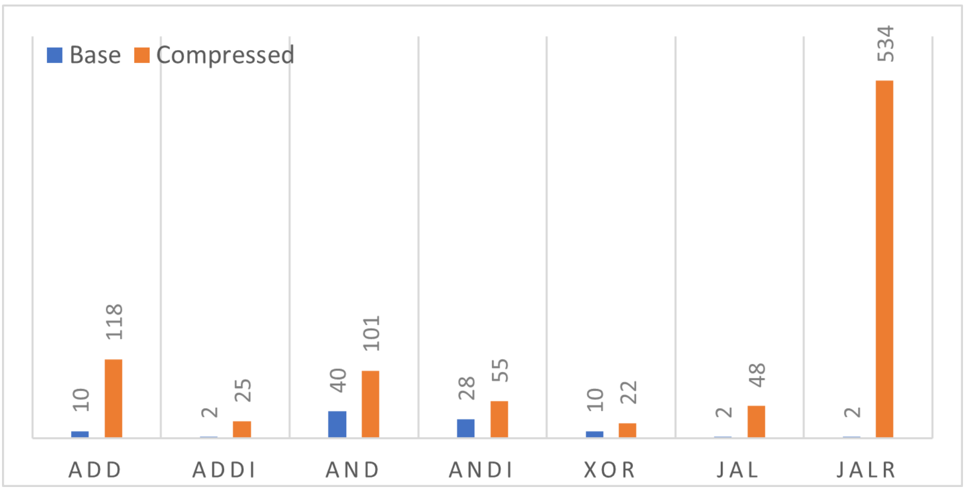


FIGURE 5.6: Comparison of DEUs (in different bits at different times) corrupting similar instructions in base form and compressed form

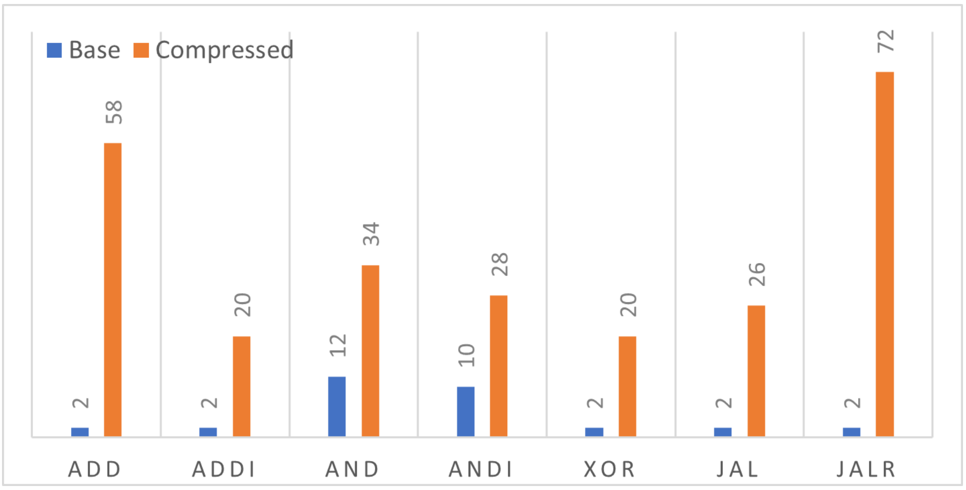


FIGURE 5.7: Comparison of DEUs (in different bits at the same time) corrupting similar instructions in base form and compressed form

instructions, or apply hardware protection to the specific structures in Table 5.7 to mitigate DEUs that may corrupt compressed logical and compressed branch instructions. On the other hand, less compressed instructions should be implemented to improve the system's reliability.

Unlike DEUs in different bits (irrespective of time), DEUs in the same bit at different times can only affect a few instructions. The affected instructions are shown in Table 5.8. To save space, only vulnerable registers and modules where crucial DEUs exist are listed. `RV32I_S*` includes `RV32I_SB`, `RV32I_SH`, and `RV32I_SW`. `RVC_*OR` includes `RVC_OR` and `RVC_XOR`. The results in Table 5.8 indicate that only a few instructions need to be enhanced to mitigate DEUs in the same bit at different times. Apart from software protection, certain hardware (registers) can also be protected to enhance these instructions.

TABLE 5.8: Vulnerable instruction in the presence of DEUs in the same bit location

Error	Register	Module
RV32I_S*	`pmp_err_q`	`load_store_unit`
RVC_*OR	`instr_rdata_alu_id_o`	`if_stage`
Breakpoint	`branch_discard_q`	`prefetch_buffer`
Ecall	`valid_q`	`FIFO`
Ecall	`rdata_q`	`FIFO`
Illegal_insn	`branch_discard_q`	`prefetch_buffer`
Hang	`valid_q`	`FIFO`
Hang	`branch_discard_q`	`prefetch_buffer`

5.7 Discussion

5.7.1 Hardware-level

In general, bits in the Ibex Core are more vulnerable to DEUs occurring in different bits (either at different times or at the same time) than in the same bits at different times. For instance, we identified a total of 13179 crucial DEUs in different bits (irrespective of time) while 18 crucial DEUs occurred in the same bits. Among crucial DEUs occurring in different bits (both at different times and at the same time), we identified more DEUs occurring at the same time (9068) compared to DEUs occurring at different times (4111). After comparing all the results, we found that the Ibex Core is most vulnerable to DEUs in different bits at the same time; less vulnerable to DEUs in different bits at different times; and even less vulnerable to DEUs in the same bits at different times. The ranking of vulnerabilities of the three groups of DEUs can determine cost-efficient protection technologies to mitigate DEU effects. For example, using the most expensive and the most effective protection technologies to protect DEUs in different bits at the same time, and using cheaper but less effective protection technologies to protect other DEUs.

We observed that DEUs in some bit/register combinations are identified multiple times. For example, most crucial DEUs occur in the combinations of registers `pmp_err_q` and `core_busy_q`; the second most crucial DEUs occur in the combinations of register `ctrl_fsm_cs` in the `controller`, `valid_q, rdata_q` in the `FIFO`, and `instr_rdata_c_id_o, rdata_outstanding_q` in the `if_stage`. Most of these combinations have direct effects on instructions in the core. In order to improve the hardware reliability in the presence of DEUs, these registers should be protected to detect/correct DEUs or be placed away to mitigate DEU occurrence possibility. Protecting these bits/registers should mitigate errors/failures caused by DEUs.

We have identified other bits that are not vulnerable to SEUs or DEUs. The DEU lists include only safe SEUs; crucial SEUs have already been identified and excluded from further consideration. That is, all the explored bits are not vulnerable to SEUs. In addition, as stated in subsection 5.5.1, the last step identifies all safe DEUs, that is, all bits that are not vulnerable to DEUs (and not vulnerable to SEUs).

Though the DEU list includes only safe SEUs, results show that these DEUs may lead to SDCs, crashes and hangs. This proves that DEUs can aggravate SEUs: though both faults are safe SEUs, the second fault can cause failures.

5.7.2 Instruction-level

In general, DEUs in different bits at different times can affect more logical and branch instructions than DEUs in different bits at the same time. Table 5.9 lists some of the instructions affected by DEUs in different bits at both different times and at the same time. 'Dt' is the number of DEUs in different bits at different times. 'St' is the number of DEUs in different bits at the same time. First of all, most DEUs at different times can corrupt more instructions than DEUs at the same time. As a result, it is important to mitigate DEUs at different times to protect the instructions. In addition, most logical instructions and branch instructions in base form tend to be less vulnerable to DEUs than the equivalent instructions in compressed form. As a result, it might be better to implement less compressed instructions in safety-critical systems. Finally, DEUs in the same bit at different times corrupt the least instructions. Most instructions are not vulnerable to these DEUs and hence need less protection.

TABLE 5.9: Comparison of instructions affected by DEUs at different times and same time

Instruction	Dt	St	Instruction	Dt	St
I_ADD	10	2	C_ADD	118	58
I_AND	40	12	C_AND	101	34
I_ANDI	28	10	C_ANDI	55	28
I_BEQ	29	6	C_BEQZ	25	20
I_OR	40	12	C_OR	20	18
I_XOR	10	2	C_XOR	22	20
I_SUB	150	24	C_SUB	108	40

DEUs in some registers play a vital role in the correct execution of instructions. The most vital registers have been discussed in Section 5.6.2. Since these registers are vital for correct instruction functionality, enhancing these registers with fault correction or placing them away to reduce DEU occurrence possibility should theoretically contribute to ensuring the functionality of instructions, and hence improving the reliability of the system.

5.8 Conclusion

In this chapter, we extended our formal method in Chapter 3 to explore DEUs. We combined model checking with fault injection to perform backward tracing of DEUs, i.e., identifying DEUs based on stating the effects of DEUs. Compared to other formal works, we further identified and classified DEUs into three four groups: safe, SDCs, crashes and hangs.

One major concern of expanding model checking to DEUs is the infeasible number of model checking runs. To solve the problem, we developed a fault-exploration strategy that enables the identification of all safe DEUs (that have no effect) in compact model checking runs. Such a strategy greatly reduces the number of model checking runs to identify all DEUs. In addition, we excluded DEUs that include crucial SEUs from the fault list. Another concern about the applicability of model checking is state explosion. We mitigated state explosion in three ways: 1) abstracting memories, 2) input constraints, and 3) focusing on architectural functionality and 4) using case splitting to reduce the input space. These complexity control strategies greatly reduced model checking runtimes and efforts. These complexity control strategies make it possible to scale model checking to DEUs without undetermined results.

We have therefore achieved all our objectives. We proved that DEUs can aggravate SEUs. For example, two safe SEUs can lead to crucial DEUs. We used our method to identify all crucial DEUs. We decreased the infeasible model checking run time of DEUs to an acceptable range. We used our method to evaluate the Ibex Core in the presence of DEUs. We found the Ibex Core is more vulnerable to DEUs in different bits (irrespective of time) than in the same bits; and more vulnerable to DEUs in different bits at the same time than in different bits at different times. We have shown that protecting against SEUs only is not enough to mitigate failures, because DEUs can aggravate SEUs.

This method can exhaustively identify vulnerable structures (such as bits, registers and modules) and instructions in the presence of DEUs at early design stages. Our method can explore all the state space; corner cases can be reached; and no testbench is required. It might be argued that since gate-level (such as DEUs are more likely to occur in adjacent bit locations) and software (some instructions are more frequent than others) play an important role, they should be considered during DEU analysis. This paper argues that the results help to determine the proper gate-level and software. The identified vulnerable structures in the presence of DEUs can contribute to place-and-route, for example, by placing these vulnerable structures in far-away locations. When developing software executed in the hardware, these identified vulnerable instructions should be avoided or enhanced.

In principle, the proposed method is compatible with other RISC-V microprocessors. There is no need to change the design or to develop formal properties and constraints. In theory, it is possible to extend the method to MEUs. It is easy to add extra fault control signals to model MEUs. After that, use the same method to explore MEUs.

There is one limitation of our method. We found that many instructions influenced by DEUs cannot be retired from the core. For example, during the second pipeline, a fetched instruction is treated as invalid due to a DEU, hence this instruction cannot be decoded or executed. In other words, this instruction is lost. However, the other instructions which are not influenced by the DEU can be executed correctly and cannot violate architectural properties. SDCs may occur due to a missing instruction. Such DEUs cannot be identified by architectural properties. Architectural properties check whether a retired instruction is executed correctly, they do not check whether an instruction is lost in a program. On the other hand, we argue that this limitation can be solved by using formal methods to explore the effects of missing instructions in a software program. This idea can be the future work of this research.

Chapter 6

Conclusion

ISO26262 suggests using simulation-based fault injection to improve safety and reliability. However, simulation-based approaches have met limitations. To test multiple faults with different benchmarks, multiple simulations are required. It is therefore impossible to use simulation-based methods to test all potential faults. Reducing the number of injected faults decreases accuracy (fault coverage). In addition, it is hard to reach some corner cases using simulation – testbenches must be carefully developed. An alternative and complementary approach is formal verification. In this research, we proposed using formal verification, mainly model checking, to exhaustively evaluate the hardware reliability in the presence of soft errors. Our method can be adapted to other RISC-V cores without developing properties and constraints again. The only modification is signal mapping. There have been many works using formal verification to perform fault analysis. However, the identified faults are only grouped into safe faults and remaining faults, and simulation-based fault injection is used to analyse the remaining faults. The main contribution of our method is to develop properties and use formal verification to further identify and categorize faults into four groups: safe faults that have no effect, and crucial faults that lead to SDCs, crashes and hangs respectively at the architectural level.

Firstly, we formalized three types of failures (SDCs, crashes and hangs) as formal properties to reveal faults that lead to SDCs, crashes and hangs respectively. In principle, the properties can be adapted to other RISC-V processors with signal remapping. We implemented a fault injection mechanism which can inject an arbitrary fault to cover the whole fault space. We developed various complexity control strategies to make sure the fault analysis is exhaustive and to improve the proof performance. We validated our properties and method with various approaches, such as simulation and mutations. We also used TMR and SR as framework validation measures. Then we performed model checking on the properties in the presence of an arbitrary SEU. Crucial SEUs can be extracted from the reported counterexamples. Safe SEUs can be found in full proofs. We developed an experimental strategy and used different formal tool settings to report determined results, there were no remaining undetermined results (except multiplications and divisions). Compared to other work that perform formal fault analysis, our method can identify and categorize all SEUs based on the SEU effects. We tested the method on a RISC-V Ibex Core written in SV. The method can explore the entire state space to find all crucial SEUs that may cause errors and failures in a reasonable time. Based on the experimental results, we can rank the reliability of all the hardware structures in the Ibex Core. We found some structures (bits, registers and modules) are more vulnerable to SEUs than others, similar to software (instructions). We found the FIFO and the Program Counter are the most vulnerable structures in the Ibex Core. In general, the Ibex Core is more vulnerable to SDCs than crashes and hangs, and the second pipeline stage contains more faults that lead to SDCs than the first pipeline stage. We found the compressed 16-bit instructions are more vulnerable to SEUs than the 32-bit instructions. We also found misaligned instructions can amplify fault effects.

Apart from evaluating the hardware reliability of the Ibex Core in the presence of soft errors at early design stages, the proposed method can also be applied to evaluate fault-tolerant technologies. Different fault-tolerant technologies hold different protection efficiency and cost. It is more cost-effective to protect the most vulnerable structures with the most efficient (but the most expensive) technology, and less vulnerable structures with less efficient (and cheaper) technology. To address the issue, we expanded the method to evaluate the fault detection effectiveness of a residue arithmetic enhanced Ibex Core. We developed properties to reveal faults (no matter safe or crucial) that can be detected by residue arithmetic. These properties also reveal structures that are detectable by residue arithmetic. We found not all the structures in the Ibex Core can be covered by residue arithmetic. It is meaningless to explore faults injected in undetectable structures, hence reducing the fault space. We then modified the properties that reveal faults that lead to failures in the last paragraph to explore whether all crucial faults can be detected by residue arithmetic. We found not all the crucial SEUs without fault-tolerant technology can be detected by residue arithmetic. However, with little hardware overhead, residue arithmetic can detect more than half of the crucial SEUs that lead to SDCs, and the detected crucial SEUs are in both the control and data paths in the Ibex Core. However, residue arithmetic is poor at detecting crucial SEUs leading to crashes and hangs. What is worse, the most vulnerable structures such as the FIFO and PC cannot be covered by residue arithmetic. The results were as expected, which proves that the method can be used to evaluate fault-tolerant technologies and to determine cost-efficient fault-tolerant technologies.

Finally, we expanded the proposed method to evaluate DEUs. One major concern of expanding model checking to DEUs is the exponential fault list, which leads to infeasible model checking runtime and state explosion. We used various approaches to solve the concern. We pruned the DEU list by removing DEUs that contain crucial SEUs. We assumed crucial SEUs will be protected, hence it is not necessary to further explore DEUs including crucial SEUs. We also compact the number of model checking runs to explore DEUs. We used multiple complexity control strategies to mitigate state explosion and to improve the proof performance. These approaches greatly reduced model checking time and effort. Together with different formal tool settings, all the reported results were determined; there were no remaining undetermined results. We exhaustively identified and categorized DEUs based on the DEU effects. We identified vulnerable hardware structures and software instructions to DEUs, and some are more vulnerable than others. We found the Ibex Core is more vulnerable to DEUs in different bits than in the same bits; and more vulnerable to DEUs in different bits at the same time than in different bits at different times. We also found that DEUs can aggravate SEUs, for example, two safe SEUs together can cause errors and failures. Hence protecting SEUs only is not enough to mitigate all errors and system failures.

We have therefore proved the hypothesis and achieved all our objectives in Chapter 1:

1. In Chapter 3, we developed a formal method to perform SEU analysis. We used TMR and SR as framework validation measures.
   (a) We developed formal properties that reveal faults leading to SDCs, crashes and hangs, and validated the properties with simulation and mutations.
   (b) We identified all SEUs, and categorized them into four groups: no effect, SDCs, crashes and hangs.
   (c) We developed complexity control strategies and experimental strategies and used different formal tool settings to ensure the exhaustiveness of the experiments: the whole state space and the fault list were covered, and there was no remaining undetermined result (except multiplications and divisions).
   (d) We evaluated the reliability of all hardware structures (bits, registers and modules) and software instruction in the presence of SEUs. We found some structures are more vulnerable to SEUs than others, similar to instructions. For example, the FIFO and the PC are the most vulnerable structures; the compressed 16-bit instructions are more vulnerable than the 32-bit instructions.
2. In Chapter 4, we expanded the developed method to formally evaluate fault-tolerant technologies in the presence of SEUs. We chose residue arithmetic as the exemplar.
   (a) We developed properties to reveal all faults that can be detected by residue arithmetic. We then modified the properties that reveal faults that lead to

failures in the last achievement to explore whether all crucial faults can be detected by residue arithmetic.

(b) We proved the above properties and identified all faults that can be detected by residue arithmetic. Hence, we found all hardware structures where injected faults can be detected by residue arithmetic. We found not all the structures in the Ibex Core can be covered by residue arithmetic. It is meaningless to explore faults in undetectable structures, hence reducing the fault space.

(c) We identified all crucial faults that can be detected by residue arithmetic. We found not all the crucial SEUs without fault-tolerant technology can be detected by residue arithmetic.

(d) We evaluated the fault detection effectiveness and hardware cost of residue arithmetic. With little hardware overhead, more than half of the bit locations in the Ibex Core are detectable by residue arithmetic. In addition, residue arithmetic can detect more than half of the crucial SEUs that lead to SDCs, including both the control and data paths in the Ibex Core. However, residue arithmetic is poor at detecting crucial SEUs leading to crashes and hangs. What is worse, the most vulnerable structures (the FIFO and PC) cannot be covered by residue arithmetic.

3. In Chapter 5, we scaled our method to DEUs.

   (a) We pruned the DEU list by removing DEUs that contain crucial SEUs. We also reduced the number of model checking runs to explore all DEUs. These approaches with the approaches in the first achievement mitigated state explosion and made the runtime and efforts feasible.

   (b) We identified and categorize all DEUs based on the fault effects. The vulnerability rank of DEUs are: DEUs in different bits at the same time > DEUs in different bits at different times > DEUs in the same bits at different times. We identified vulnerable hardware structures and software instructions to DEUs. All the results were determined (except multiplications and divisions).

   (c) We evaluated the reliability of all hardware structures and software instructions in the presence of DEUs. We found some structures (such as the FIFO) and instructions (such as the compressed instructions) are more vulnerable than others. Another example is that logical and branch instructions are more vulnerable to DEUs in different bits at different times.

   (d) We proved that DEUs can aggravate SEUs, for example, two safe SEUs together can cause errors and failures. Hence protecting SEUs only is not enough to mitigate all errors and system failures.

4. In Chapter 3, 4 and 5, we demonstrated the above proposed methods in a RISC-V Ibex Core.

One application of our method is to exhaustively evaluate the hardware reliability against SEUs or DEUs at early design stages. We have demonstrated this application. The proposed method overcomes the limitations of simulation-based fault injection, and would, in principle, be compatible with ISO26262 Another application of the method is to determine cost-efficient hardware and software protection technologies at early design stages. The method can find vulnerable hardware structures and vulnerable software instructions to choose hardware and software protection technologies. The method can also evaluate fault-tolerant technologies.

Though formal verification has advantages listed in Chapter 2, there are several limitations of formal verification (especially model checking). Formal verification knowledge is required to develop formal properties, which is more difficult than developing testbenches. In addition, the correctness of formal verification depends on the design and verification specifications. For example, in this research, if an incorrect design specification (such as an incorrect Ibex guide and RISC-V Manuals) was provided, wrong properties and incomplete assumptions would be developed, which would lead to incorrect formal verification results. Formal verification is also limited by design complexity. The state space to be explored explodes exponentially as the design size increases. It is too computationally intensive and time-consuming to verify complex designs. Moreover, verification of IPs that are prohibited from access during verification is a challenge in formal verification.

One limitation of this research is not exploring two scenarios of hangs: Dead State and Live State described in Section 3.3. Fully proving liveness is extremely difficult using model checking. Another limitation is that the architectural properties can only verify retired instructions. If an instruction cannot be retired from the core due to faults, for instance, a pipeline flush, this instruction is lost. Missing instructions in a software program can lead to SDCs. However, SDCs caused by missing instructions in the software program cannot be captured by architectural properties. Architectural properties verify whether a retired instruction was executed correctly, they do not check whether an instruction was lost. One possible solution is to use strobe properties, but strobe properties can produce false negatives, for example, if the faulty core took more clock cycles than the golden core to produce correct outputs, strobe properties can fail. An alternative is using formal methods to explore the effects of missing instructions in a software program. This proposal might be the future work of this research.

Appendix A

Brief SytemVerilog Assertion Syntax

```
##n;
##[2:$];
```

##*n* delays n clock cycles. ##[2 : $] delays 2 to infinite clock cycles.

```
|->
|=>
```

The LHS of the implication operator is called the antecedent and the RHS is called the consequent. If $|- >$ is used, the consequent is evaluated at the same clock cycle if the antecedent is true. If $| =>$ is used, the consequent is evaluated at the next clock cycle if the antecedent is true. $| =>$ is equivalent to $|- >$ ##1.

```
Label:
assume property (property_expr);
```

The *property_expr* in the ′assume′ is used to limit the behaviour of DUT inputs. Also known as constraints. The syntax of ′assert′ and ′cover′ is similar. ′Assert′ checks that the property holds under all circumstances. ′Cover′ tries to demonstrate one example of how the sequence can be completed given the design and assumptions.

```
$past(expr, N);
```

Returns the value of *expr* N clock cycles ago.

```
$rose(expr);
```

Returns TRUE if *expr* is TRUE in this cycle and was FALSE in the previous cycle, otherwise returns FALSE.

```
$fell(expr);
```

Returns TRUE if *expr* is FALSE in this cycle and was TRUE in the previous cycle, otherwise returns FALSE.

```
$stable(expr);
```

Returns TRUE if *expr* has the same value this cycle as it did in the previous cycle, otherwise returns FALSE. We used it in 'assume' to constrain *expr* cannot change during model checking; without it, *expr* may change every clock cycle.

```
$onehot(expr);
```

Returns TRUE if *expr* has exactly one bit with the value $1'b1$, otherwise returns FALSE. We wire all XOR gate trigger signals to *expr*. Then we used this function in 'assume' to make sure only one XOR gate is triggered (only one fault is injected) during model checking.

```
disable iff (expr)
```

If *expr* is TRUE then all current outstanding obligations for that property (including all overlapping ones) are removed. For example, if the DUT is under reset condition, then the property should not be checked.

Appendix B

Descriptions of Jasper proof engines used

Engine Hps focuses on finding proofs. It remembers unsuccessful proof attempts and skips them when revisiting a property factor loop. This engine does not skip proof attempts if additional invariants or proven directives are applied. Hp is the multi-property version of engine Hps.

Engine Hts focuses on finding counterexamples. Ht is the multi-property version of engine Hts.

Engine B uses SAT solvers and abstractions to find counterexamples. This engine is similar to engine Ht but can outperform engine Ht in some cases. However, this engine can never find an exhaustive proof. It can only give counterexamples or bounded proofs.

Engine D proves or finds counterexamples for one property at a time. It uses on-the-fly compression of the proof for increased capacity with deeper proofs.

Engine I proves or finds a counterexample for one property at a time, which is similar to Engine D. Engine I iteratively includes logic from the COI, thus minimizing the amount of logic necessary for a proof. This engine can be used in combination with engines C, C2, K, and N to speed up the proof process. By default, engine I exchanges information with engines C, K, and N during the proofs of the same property to speed up the process.

Engines C and C2 prove or find counterexamples for one property at a time. These engines are often better for verifying complex sequential properties such as for datapath credit or token management units. They use abstractions to iteratively include logic from the COI, thus minimizing the amount of logic necessary for a proof.

Engine K is optimized to find bounded proofs. This engine only searches for traces, and it will generally not find proofs. It uses abstractions to iteratively includes logic from the COI, minimizing the amount of logic used while searching.

Engine L is a bug-hunting engine and focuses on finding counterexamples or hitting cover points that are normally hard to reach by conventional formal engines. It is based on a combination of bounded model checking algorithms and state selection heuristics for smart traversal of a subset of the reachable state space.

Engine M can find full proofs and is not limited to finding traces like engines B, K, and L. The focus of this engine is on proving properties that are valid (assertions) or unreachable (covers). It works well on liveness properties. This engine works best on properties with a small COI and without any complex constraints.

Engine N can find full proofs and is not limited to finding traces like engines B, K, and L. This engine works best on properties with a small COI and without any complex constraints. Engine N can be used in combination with engines C, I, and K to speed up the proof process. By default, engine N exchanges information with engines C, I, and K during the proofs of the same property to speed up the process.

Engine R is multi-property engine that focuses on proofs. This engine is better at finding proofs (like Hp) than traces and it provides trace attempts, proof attempts, and min_length updates. This engine uses a proof strategy similar to engine M. In addition, it limits resources with a timeout and automatically restarts with longer timeouts.

Engine Tri uses several processes at once, to process one property. During the proof, it gives proof attempt messages while computing the reachable states.

Engine U is a SAT-based multi-property engine that uses constraint solving to walk along the set of reachable states of the given model. It makes an effort to choose the next state to visit uniformly at random from the set of all available next states.

Abstraction Engines AM and AD follow the same algorithm of their non-abstraction counterparts, engines M and D. One difference is that Engines AM and AD start with a small number of flops/gates in the analysis region and gradually add more. For large designs, where a sequentially deep bound is needed to get a precise abstraction, A engines might be better than other abstraction engines, such as K, C, I, and N. A engines might be especially helpful for problems that have a big COI but a small proof witness.

Appendix C

Using Formal Methods to Evaluate Hardware Reliability in the Presence of Soft Errors

Using Formal Methods to Evaluate Hardware Reliability in the Presence of Soft Errors

Bing Xue and Mark Zwolinski
School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, UK
bx1u17@soton.ac.uk, mz@ecs.soton.ac.uk

***Abstract*—Reliability is a major concern in many embedded systems. Redundancy-based methods are widely used against Single Event Upsets, causing significant temporal and spatial overhead. The traditional method to evaluate the reliability of a system is fault injection. However, it is practically impossible to test all faults for a complex design due to intractable simulation times. In this paper, we propose using formal methods to evaluate hardware reliability in the presence of soft errors. The proposed method can exhaustively search the entire state space and the whole fault list in a reasonable time. The method is applied to assess the vulnerability of all registers in a RISC-V Ibex core.**

***Index Terms*—Formal verification, soft errors, single event upset, fault injection, hardware reliability.**

I. Introduction

EMBEDDED systems are widely employed in electrical and mechanical devices. Unreliable embedded systems may lead to severe consequences. Single Event Upsets, also called soft errors, have been a major concern for electronic reliability in embedded systems [1], [2]. The traditional technique to evaluate the hardware reliability is fault injection and simulation. ISO 26262 requires (random) fault injection to improve safety and reliability [3]. However, the biggest limitation of fault injection is the simulation time. It is theoretically possible but practically impossible to test all faults using fault injection.

This research proposes a method that uses formal verification to evaluate the hardware reliability of a microprocessor in the presence of soft errors. Traditional approaches are forward-tracing: injecting faults and monitoring response. We combined formal verification with fault injection to perform backward-tracing: searching all candidate faults for a given error. The proposed method can exhaustively search the whole state space to find all candidate faults that may cause silent data corruption, crashes and hangs in a reasonable time. We used this method to assess the vulnerability of all register bits in a RISC-V Ibex core.

This paper is organized as follows. Section 2 introduces some terminology used in this paper and reviews some related works. Sections 3 and 4 present the methods and results of this methodology. Section 5 has the evaluation of the method and Section 6 is the conclusion.

II. Background and Related Works

Single event upsets (SEUs) have been a major concern in commercial terrestrial digital circuits [4]. An SEU is a soft error because it does not permanently damage hardware. The effects of such faults may be significant, but others will have no effect. Based on past research [5]–[7], the effects of SEUs on microprocessors can be categorized as follows:

- Correct: The program completes normally. The output, the execution time and the internal states match the golden run.
- Silent Data Corruption (SDC): The program completes normally without indication of errors, but the output differs from the golden run.
- Crash: The program crashes, an error indication arises.
- Hang: The program fails to complete in a limited number of clock cycles.

The Ibex core is an open-source 32-bit RISC-V 2-stage CPU core [8]. Figure 1 shows the Ibex pipeline [8]. The first pipeline stage fetches instructions from instruction memory and decompresses compressed instructions. The second pipeline stage decodes and executes fetched instructions, reads from and writes to the register file. There are 71 registers (2008 bits) in the Ibex core. This research focuses on the registers in the core; Instruction Memory and Data Memory are not considered.

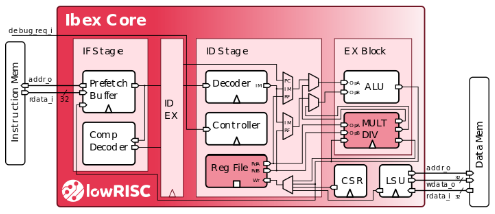


Fig. 1. Ibex Pipeline from [8]

Formal verification uses mathematical methodologies to verify whether the design under analysis meets the design specification [9]. Model checking is a formal method to analyse dynamic systems that can be modelled by state-transitions [10]. Model checking uses algorithms, such as assertion-based algorithms, to exhaustively explore all reachable states in the design to analyse if the design meets the formal specification [10]. If not, a counterexample will be given.

Formal verification (FV) has been used for verification and validation of processor cores. Gao used FV to validate a CHERI-enabled processor [11]. Lonsing used FV to prove the soundness of a technique called SQED [12]. Jakobs verified processors with custom instruction set extensions [13]. Rojas verified RISC-V processors described in Chisel [14].

Formal verification can also be used to evaluate reliability. Silva proposed an approach to identify functionally safe faults [15]. FV generates a Cone of Influence (COI) for a target. Faults outside the COI cannot affect the target, hence are safe faults. COI analysis is also used in this research to find structurally safe faults. Jayakumar modelled faulty states/outputs as assumptions and used Simulink Design Verifier to find faults that meet the assumptions [16]. However, complex properties and assumptions may lead to a state space explosion. In contrast to Jayakumar [16], we have modelled correct behaviours as properties. In addition, memories are abstracted to mitigate state explosion.

III. Formal Method to Evaluate Reliability

We propose using a formal method to evaluate hardware reliability in the presence of SEUs. This method uses model checking to find all the SEUs that cause errors and failures. SystemVerilog Assertions (SVA) are used as the formal language. Cadence JasperGold is used as the formal tool. There are four steps in the method:

- Implement a fault injection mechanism.
- Develop properties that can search and categorize faults.
- Abstract memories and develop constraints.
- Run model checking and COI analysis in parallel.

A. Fault Injection Mechanism

The first step is to implement the fault injection mechanism in the Ibex core. SVA strobe properties are developed to validate the mechanism has no impact on the Ibex core. Strobe properties compare important signals (such as instructions) in two Ibex cores: one golden and one with SEUs.

```
assert_rvfi_insn: assert property (
 @(posedge clk_i) disable iff (!rst_ni)
 golden_rvfi_valid && faulty_rvfi_valid |->
 faulty_rvfi_insn == golden_rvfi_insn );
```

The above assertion compares the retired instructions from the two Ibex cores. When both cores finish executing a valid instruction, `golden_rvfi_valid` and `faulty_rvfi_valid` are set for one clock cycle. The retired instructions are stored in register `golden_rvfi_insn` and register `faulty_rvfi_insn`. If the assertion fails, the injected fault has changed the executed instruction, causing a system failure.

B. Properties to search Faults

The second step is to develop properties that can find faults according to the effects of faults. Different properties are developed to find faults that cause SDC, crash and hang. These developed properties are validated using model checking.

1) SDC: When an SDC occurs, the behaviour or the outputs of the design differs from the golden design. Multiple properties can detect such failure. For example, strobe properties described above and architectural properties that define the correct behaviour of each instruction. After comparison, both properties produce the same results, though strobe properties are simpler and faster. As a result, strobe properties are used to find faults causing SDCs.

2) Crash: For this research, we have disabled all the interrupts in the Ibex core. Ibex operates in the machine mode. Only exceptions in the machine mode can cause crashes. When a crash is encountered, the exception code is stored in a Control Status Register (CSR). Crash properties are developed according to the exception code.

The following is an example of a store access fault assertion. `crash_priv_mode` is the current privilege mode; `2'b11` means machine mode. `crash_mcause_q` reads the value in control status register mcause. Assertion `assert_store_access_fault` should never fail for the golden core. If a fault would cause a crash, the assertion fails.

```
assert_store_access_fault: assert property (
 @(posedge clk_i) disable iff (!rst_ni)
 (crash_priv_mode==2'b11) |->
 (crash_mcause_q!=6'd7) );
```

3) Hang: There are three possible scenarios for a hang:

- WFI: The retired instruction is Wait-For-Instruction. Since all interrupts are disabled, the core stays in the sleep state.
- Dead State: The core can be thought of as a Finite State Machine (FSM). The FSM is stuck in a state and cannot leave that state.
- Live State: The FSM is stuck in a state sequence. The FSM cannot return to a specific state from other states.

JasperGold Superlint can generate properties for the last two scenarios. However, these properties are liveness properties. A liveness property describes something good that eventually happens. For example, the FSM can eventually return to a state from other states. Liveness should be used with caution because it may be time-consuming to prove liveness properties. As a result, only the first scenario (WFI) is modelled.

C. Memory Abstraction and Constraints

As shown in Figure 1, there are two memories that store instructions (IMEM) and data (DMEM). Including memories in formal verification increases the complexity, which may lead to a state space exposition. As a result, IMEM and DMEM are abstracted. In addition, constraints are developed using assumptions to speed up model checking and avoid false negatives. It is assumed that one clock cycle after receiving an instruction/data request, the abstracted memories output an instruction/data with a grant signal. Instructions from IMEM are constrained to legal RV32IMC instructions.

D. Model Checking and COI Analysis

Model checking is performed to find all faults that may cause SDC/crash/hang in each bit. A script was developed to automatically run model checking using the JasperGold FPV app. COI Analysis is performed in parallel. JasperGold can automatically generate a COI for each property. A fault that is outside the property COI is a structurally safe fault. Safe faults will never cause a corresponding assertion failure. By running structural analysis in JasperGold, 81 structurally safe faults that have no effect were found.

IV. Results and Analysis

After model checking, there are three possible statuses for each property: Proven, Bounded Proven and Failure. If the status of a property is 'Proven', the property is proved. A fault in the corresponding bit cannot cause errors that violate the property. As a result, the bit is not vulnerable to errors. If the status of a property is 'Failure', there is a counterexample for the property. A fault in the corresponding bit may cause errors that violate the property. As a result, the bit is vulnerable to errors.

State explosion is a problem in model checking. Bounded model checking is used as an alternative. If the status of a property is 'Bounded Proven', the formal tool cannot prove or disprove the property within a bounded trace or time. If there exists a counterexample, it is beyond the maximum trace or time limit. In other words, it is hard to find a fault in the bit that can fail the property assertion. The bit is less reliable against errors.

Based on the reliability of faults, faults can be classified into safe faults (Proven), vulnerable faults (Failure) that can cause SDC/crash/hang, and less reliable faults (Bounded Proven) that are less likely to cause SDC/crash/hang.

1) SDC: Table I shows results of checking strobe properties for faults causing SDC. There are 81 common safe bits. The common safe bits are from COI analyses. All the safe bits are not consumed (i.e. further used) in the design. For example, register stored_addr_q stores a request address to memory. Since the address is aligned by 4, the two least significant bits are not consumed, hence are safe.

TABLE I
Strobe Properties for RV32IMC

Name	Proven	Bounded Proven	Failure
Instruction_is_done	81	689	1238
Instruction	81	675	1252
rs1_address	81	691	1236
rs2_address	81	693	1235
rd_address	81	690	1237
rs1_read_data	81	692	1235
rs2_read_data	81	693	1234
rd_write_data	81	689	1238
current_PC	81	690	1237
next_PC	81	689	1238
memory_address	81	693	1234
memory_read_data	81	710	1217
memory_read_mask	81	739	1188
memory_write_data	81	704	1223
memory_write_mask	81	736	1191

Apart from safe faults from COI analysis, there are no Proven results. Proving strobe properties is time-consuming. Even without any faults, it takes more than 48 hours to get undetermined results. To save time, bounded model checking is used.

It was found that there are common faults that can cause all types of assertion failures. These faults include faults in the register file, though not every bit in register file is included. These common faults causing multiple types of SDC are more dangerous than other faults causing a single SDC.

2) Crash: Table II shows results of crash properties with legal RV32IMC instructions. Most faults cannot cause a corresponding crash. From the COI analysis, there are 82 common structural safe bits, which are included in the Proven results. Compared to 81 safe bits that cannot cause a SDC, the extra safe bit is instr_new_id_q. This bit is used to capture signals listed in Table I. In the real Ibex core, this bit is assigned to a value but never used. Although instr_new_id_q influences signals used for formal verification, it does not influence the behaviour of the Ibex core. Hence it is a safe bit.

TABLE II
Crash Properties for RV32IMC

Name	Proven	Bounded Proven	Failure
Insn_access_fault	2002	0	6
Illegal_insn	1908	5	95
breakpoint	1963	2	43
load_access_fault	2006	0	2
store_access_fault	2006	0	2
ECall_MMode	2004	0	4

Faults in some bits can cause multiple crashes. For example, there are five bits where faults in these bits can cause an `ECall_MMode`, breakpoint and `Illegal_insn`. Faults in two bits can cause `load_access_fault` and `store_access_fault`. Faults that may cause a breakpoint may also cause `Illegal_insn`. Bits with faults causing multiple crashes are more vulnerable than bits with faults causing a single crash.

3) Hang: Results show that faults in 1995 bits cannot cause a hang (WFI); faults in 10 bits can cause a hang (WFI); and faults in 3 bits are Bounded Proven.

One notable finding is that an incorrect sequence of instructions may cause a hang. The following is an example of the assembly code of a for-loop:

```
loop:
   BEQZ x1, loopend     # if i == 0, break loop
   SUB x1, x1, 1        # i = i - 1
   JAl x0, loop         # jump to loop
loopend:
   ADDI x3, x1, 0       # a = i
```

In the above assembly code, if the value in register x1 is zero, the for-loop ends with the instruction BEQZ. When x1 is zero, if there is a fault when reading or executing BEQZ, then the for-loop will never end, causing a hang. Several faults can cause such a problem. For example: faults causing an incorrect instruction address such that BEQZ is not either fetched from FIFO or read from memory; or wrong results (faults in register file) of BEQZ. We found both software and hardware lead to a hang. To find all faults causing hangs, further research is needed.

Faults that cause errors under all conditions are not found. Faults in one bit may cause multiple types of errors. The more types of errors a fault may cause, the more vulnerable the bit is. We define the vulnerability of each bit as

$(NumberFailures) \times 4 + (NumberBoundedProven) \times 2$

Thus the reliability of all bits can be calculated and ranked.

V. Evaluation

The proposed method can successfully evaluate the hardware reliability in the presence SEUs. We used formal verification to explore the entire state space and the whole fault list to search all faults causing SDC/crash/hang. It is found that there are 81 bits where faults in these bits cannot cause SDC/crash/hang. We used this method to rank the reliability of all 2008 bits in the Ibex core. It is found that some bits are more vulnerable to SEUs than others. For example, the 1-bit register `instr_valid_id_q` is very vulnerable because faults in this bit can cause all types of SDC and four types of crash. In addition, some registers are only vulnerable to certain failures. For example, `register ctrl_fsm_cs` is only vulnerable to SDC.

Apart from faults, one notable finding is that an inappropriate sequence of instructions caused by poor compilation can lead to system failures. For example, the jump address of branch instructions must be aligned by four. Otherwise, the Ibex core will try to align instruction stored in the jump address by reversing the upper 16 bits with the lower 16 bits. In some cases, reversed instructions are illegal, causing illegal_insn exception even without any faults. Similar problems exist in other properties, such as the for-loop example in the last section. This kind of problem should never arise with a good compiler.

There are limitations to the current work. For instance, state explosion and failing to find all faults causing hangs (Dead State and Live State). The next phase of this research will focus on finding faults that cause hangs with both hardware and software. In addition, we will look at how to mitigate the state explosion. After that, an efficient methodology to mitigate against SEUs based on the rank of their reliability will be developed. The protection methodology will keep a balance between overhead and efficiency. The more vulnerable the register is, the more expensive but more efficient protection technology will be applied. The protection methodology will contain two protection strategies:

- Assuming the sequence of instructions is arbitrary, users can do whatever they want. Extra protection is necessary against poor compilation.
- Assuming a qualified compiler is used. No protection against poor compilation. This is cheaper than strategy one.

VI. Conclusion

One major concern of embedded systems is Single Event Upsets. Single Event Upsets, also called soft errors, can cause system failure or nothing. This research proposes a formal method that can explore the whole state space and the whole fault list to search faults that may cause nothing/SDC/crash/hang. This method can successfully evaluate the SEU reliability of all bits in the RISC-V Ibex core. Results indicate that there are three types of bits: safe bits, vulnerable bits, and less reliable bits. Faults in safe bits cannot cause SDC/crash/hang, faults in vulnerable bits can cause SDC/crash/hang, faults in less reliable bits are unlikely to cause SDC/crash/hang. There is no fault that can always cause SDC/crash/hang. Compared to fault injection, the advantages of this method are exhaustive search and short time. The limitations of the current work are state explosion and liveness properties. The next part of this research will focus on improving the limitations and developing an efficient protection methodology against SEUs.

References

[1] A. Vijayan, S. Kiamehr, M. Ebrahimi, K. Chakrabarty, and M. B. Tahoori, "Online soft-error vulnerability estimation for memory arrays and logic cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 2, pp. 499–511, Feb 2018.

[2] R. Travessini, P. R. C. Villa, F. L. Vargas, and E. A. Bezerra, "Processor core profiling for seu effect analysis," in *2018 IEEE 19th Latin-American Test Symposium (LATS)*, March 2018, pp. 1–6.

[3] *ISO 26262-1: Road vehicles — Functional safety*, ISO Std.

[4] A. Dixit and A. Wood, "The impact of new technology on soft error rates," in *2011 International Reliability Physics Symposium*, April 2011, pp. 5B.4.1–5B.4.7.

[5] A. Ramos, J. A. Maestro, and P. Reviriego, "Characterizing a risc-v sram-based fpga implementation against single event upsets using fault injection," *Microelectronics Reliability*, vol. 78, pp. 205 – 211, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0026271417304493

[6] H. Cho, "Impact of microarchitectural differences of risc-v processor cores on soft error effects," *IEEE Access*, vol. 6, pp. 41 302–41 313, 2018.

[7] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2017, pp. 97–108.

[8] lowRISC. Ibex: An embedded 32 bit risc-v cpu core. [Online]. Available: https://github.com/lowRISC/ibex

[9] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: A survey," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 4, no. 2, pp. 123–193, Apr. 1999. [Online]. Available: http://doi.acm.org/10.1145/307988.307989

[10] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of model checking*. Springer, 2018, vol. 10.

[11] D. Gao and T. Melham, "End-to-end formal verification of a risc-v processor extended with capability pointers," in *2021 Formal Methods in Computer Aided Design (FMCAD)*, 2021, pp. 24–33.

[12] F. Lonsing, S. Mitra, and C. Barrett, "A theoretical framework for symbolic quick error detection," in *2020 Formal Methods in Computer Aided Design (FMCAD)*, 2020, pp. 1–10.

[13] M.-C. Jakobs, F. Pauck, M. Platzner, H. Wehrheim, and T. Wiersema, "Software/hardware co-verification for custom instruction set processors," *IEEE Access*, pp. 1–1, 2021.

[14] C. Rojas, H. Morales, and E. Roa, "A low-cost bug hunting verification methodology for risc-v-based processors," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[15] F. A. da Silva, A. C. Bagbaba, S. Sartoni, R. Cantoro, M. S. Reorda, S. Hamdioui, and C. Sauer, "Determined-safe faults identification: A step towards iso26262 hardware compliant designs," in *2020 IEEE European Test Symposium (ETS)*, 2020, pp. 1–6.

[16] A. V. Jayakumar and C. Elks, "Property-based fault injection: A novel approach to model-based fault injection for safety critical systems," in *Model-Based Safety and Assessment*, M. Zeller and K. Höfig, Eds. Cham: Springer International Publishing, 2020, pp. 115–129.

Appendix D

Fault Injection Control Module and XOR gates

The following is the FI controller to inject a single SEU. To save space, some code is replaced with comments.

```
module FI_control (
    input  logic                               clk_i,
    input  logic                               rst_ni,

    // injection time
    input  logic [31:0]                        FI_time,
    // injection bit index
    input  logic [11:0]                        FI_index,

    // Mask Signals
    // ibex_controller
    output logic [3:0]                        FI_ctrl_fsm_cs,
    output logic                              FI_nmi_mode_q,
    output logic                              FI_do_single_step_q,
    output logic                              FI_debug_mode_q,
    output logic                              FI_enter_debug_mode_prio_q,
    output logic                              FI_load_err_q,
    output logic                              FI_store_err_q,
    output logic                              FI_exc_req_q,
    output logic                              FI_illegal_insn_q,

    // ibex_cs_registers
    output logic [1:0]                        FI_priv_lvl_q,
    output logic [2:0]                        FI_mcountinhibit_q,
    output logic [5:0]                   FI_mstatus_q,
    output logic [31:0]                  FI_mepc_q,
    output logic [17:0]                  FI_mie_q,
    output logic [31:0]                  FI_mscratch_q,
    output logic [5:0]                   FI_mcause_q,
    output logic [31:0]                  FI_mtval_q,
    output logic [31:0]                  FI_mtvec_q,
    output logic [31:0]                  FI_dcsr_q,
    output logic [31:0]                  FI_depc_q,
    output logic [31:0]                  FI_dscratch0_q,
    output logic [31:0]                  FI_dscratch1_q,
    output logic [2:0]                   FI_mstack_q,
    output logic [31:0]                  FI_mstack_epc_q,
    output logic [5:0]                   FI_mstack_cause_q,
    output logic [5:0]                   FI_cpuctrl_q,
    output logic [63:0]                  FI_mcycle,
    output logic [63:0]                  FI_minstret,

    // ibex_fetch_fifo
    output logic [30:0]                  FI_instr_addr_q,
    output logic [2:0]                   FI_valid_q,
    output logic [31:0]                  FI_rdata_q [3],
    output logic [2:0]                   FI_err_q,

    // ibex_id_stage
    output logic [33:0]                  FI_imd_val_q[2],
    output logic                         FI_branch_set_raw_q,
    output logic                         FI_branch_jump_set_done_q,
    output logic                         FI_id_fsm_q,

    // ibex_if_stage
    output logic                         FI_instr_valid_id_q,
    output logic                         FI_instr_new_id_q,
    output logic [31:0]                  FI_instr_rdata_id_o,
    output logic [31:0]                  FI_instr_rdata_alu_id_o,
    output logic                         FI_instr_fetch_err_o,
    output logic                         FI_instr_fetch_err_plus2_o,
    output logic [15:0]                  FI_instr_rdata_c_id_o,
    output logic                         FI_instr_is_compressed_id_o,
    output logic                         FI_illegal_c_insn_id_o,
    output logic [31:0]                  FI_pc_id_o,
    // ibex_load_store_unit
    output logic [23:0]                 FI_rdata_q_ls,
    output logic [1:0]                  FI_rdata_offset_q,
    output logic [1:0]                  FI_data_type_q,
    output logic                        FI_data_sign_ext_q,
    output logic                        FI_data_we_q,
    output logic [31:0]                 FI_addr_last_q,
    output logic [2:0]                  FI_ls_fsm_cs,
    output logic                        FI_handle_misaligned_q,
    output logic                        FI_pmp_err_q,
    output logic                        FI_lsu_err_q,

    // ibex_multdiv_fast
    output logic [4:0]                  FI_div_counter_q,
    output logic [2:0]                  FI_md_state_q,
    output logic [31:0]                 FI_op_numerator_q,
    output logic [31:0]                 FI_op_quotient_q,
    output logic                        FI_div_by_zero_q,
    output logic [1:0]                  FI_mult_state_q,

    // ibex_prefetch_buffer
    output logic [31:0]                 FI_stored_addr_q,
    output logic [31:0]                 FI_fetch_addr_q,
    output logic                        FI_valid_req_q,
    output logic                        FI_discard_req_q,
    output logic [1:0]                  FI_rdata_outstanding_q,
    output logic [1:0]                  FI_branch_discard_q,
    output logic [1:0]                  FI_rdata_pmp_err_q,

    // ibex_register_file_ff
    output logic [31:0]                 FI_rf_reg_q[31],

    // ibex_top
    output logic                        FI_core_busy_q
);

  logic [2007:0] mask_onehot;
  assign {/*all the 2008 bits of the output Mask Signals*/} = mask_onehot;

  //A simple counter is not shown to save space
  wire flag;
  assign flag = (counter == FI_time);

  always_comb begin
    onehot_control = '0;
    if (flag) begin
      onehot_control [FI_index] = 1'b1;
    end
  end

endmodule
```

The following is implementing an XOR gate to inject an SEU.

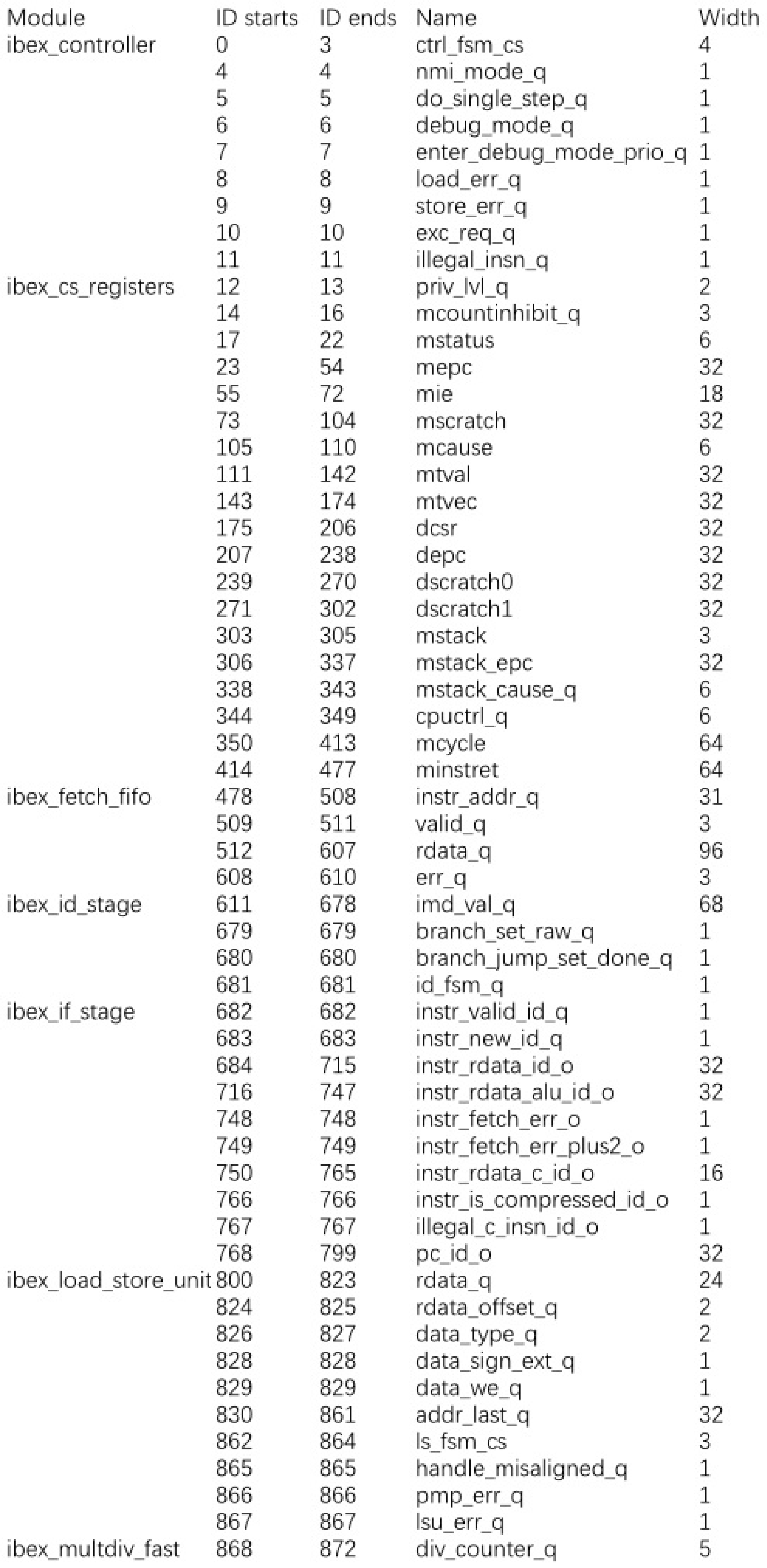
```
  // Before implementing XOR
  valid_q <= valid_d;

  // After implementing XOR
  valid_q <= valid_d ^ FI_valid_q;
```

The following is the modified FI controller to inject DEUs. To save space, some repeated lines from the above are not shown.

```
module FI_control (
    input  logic                        clk_i,
    input  logic                        rst_ni,

    // injection time
    input  logic [31:0]                 FI_time1, FI_time2,
    // injection bit indexes
    input  logic [11:0]                 FI_index1, FI_index2,

    // Mask Signals
    // repeated lines
    output logic                       FI_core_busy_q
);

  logic [2007:0] mask_onehot;
  // repeated lines

  wire flag1, flag2;
  assign flag1 = (counter == FI_time1);
  assign flag2 = (counter == FI_time2);

  always_comb begin
    onehot_control = '0;
    if (flag1) begin //flag1 sets if the time meets the first fault inject
      onehot_control [FI_index1] = 1'b1;
    end
    `ifdef DEU //If double fault injection is enabled
    if (flag2) begin //flag2 sets if the time meets the second fault injec
      onehot_control [FI_index2] = 1'b1;
    end
    `endif
  end

endmodule
```

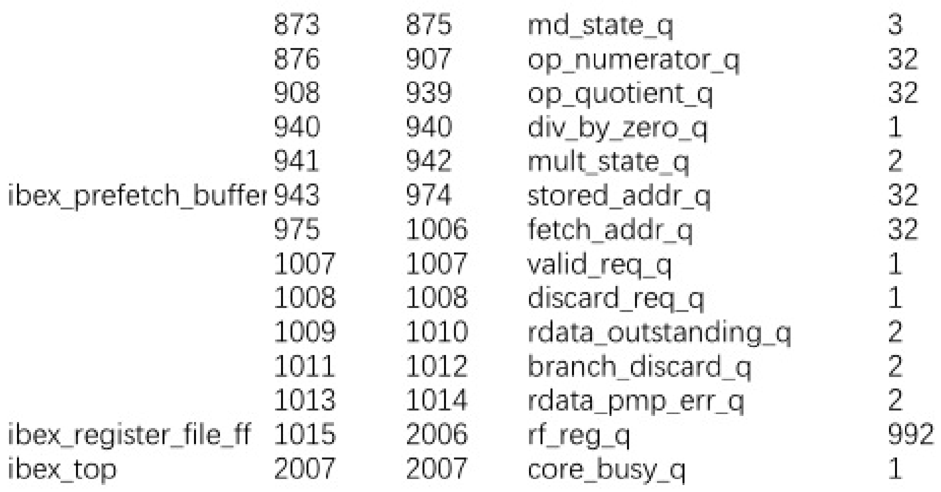
Appendix E

Bit ID in the Ibex Core



Module	ID starts	ID ends	Name	Width
ibex_controller	0	3	ctrl_fsm_cs	4
	4	4	nmi_mode_q	1
	5	5	do_single_step_q	1
	6	6	debug_mode_q	1
	7	7	enter_debug_mode_prio_q	1
	8	8	load_err_q	1
	9	9	store_err_q	1
	10	10	exc_req_q	1
	11	11	illegal_insn_q	1
ibex_cs_registers	12	13	priv_lvl_q	2
	14	16	mcountinhibit_q	3
	17	22	mstatus	6
	23	54	mepc	32
	55	72	mie	18
	73	104	mscratch	32
	105	110	mcause	6
	111	142	mtval	32
	143	174	mtvec	32
	175	206	dcsr	32
	207	238	depc	32
	239	270	dscratch0	32
	271	302	dscratch1	32
	303	305	mstack	3
	306	337	mstack_epc	32
	338	343	mstack_cause_q	6
	344	349	cpuctrl_q	6
	350	413	mcycle	64
	414	477	minstret	64
ibex_fetch_fifo	478	508	instr_addr_q	31
	509	511	valid_q	3
	512	607	rdata_q	96
	608	610	err_q	3
ibex_id_stage	611	678	imd_val_q	68
	679	679	branch_set_raw_q	1
	680	680	branch_jump_set_done_q	1
	681	681	id_fsm_q	1
ibex_if_stage	682	682	instr_valid_id_q	1
	683	683	instr_new_id_q	1
	684	715	instr_rdata_id_o	32
	716	747	instr_rdata_alu_id_o	32
	748	748	instr_fetch_err_o	1
	749	749	instr_fetch_err_plus2_o	1
	750	765	instr_rdata_c_id_o	16
	766	766	instr_is_compressed_id_o	1
	767	767	illegal_c_insn_id_o	1
	768	799	pc_id_o	32
ibex_load_store_unit	800	823	rdata_q	24
	824	825	rdata_offset_q	2
	826	827	data_type_q	2
	828	828	data_sign_ext_q	1
	829	829	data_we_q	1
	830	861	addr_last_q	32
	862	864	ls_fsm_cs	3
	865	865	handle_misaligned_q	1
	866	866	pmp_err_q	1
	867	867	lsu_err_q	1
ibex_multdiv_fast	868	872	div_counter_q	5

FIGURE E.1: The first part of bit IDs



	873	875	md_state_q	3
	876	907	op_numerator_q	32
	908	939	op_quotient_q	32
	940	940	div_by_zero_q	1
	941	942	mult_state_q	2
ibex_prefetch_buffer	943	974	stored_addr_q	32
	975	1006	fetch_addr_q	32
	1007	1007	valid_req_q	1
	1008	1008	discard_req_q	1
	1009	1010	rdata_outstanding_q	2
	1011	1012	branch_discard_q	2
	1013	1014	rdata_pmp_err_q	2
ibex_register_file_ff	1015	2006	rf_reg_q	992
ibex_top	2007	2007	core_busy_q	1

FIGURE E.2: The last part of bit IDs

Appendix F

Bit patterns of valid RV32IMC instructions

```
logic is_rv32_insn;
assign is_rv32_insn = (instr_rdata_i[1:0] == 2'b11);

logic RV32I_LUI, RV32I_AUIPC, RV32I_JAL, RV32I_JALR,
 RV32I_BEQ, RV32I_BNE, RV32I_BLT, RV32I_BGE, RV32I_BLTU,
 RV32I_BGEU, RV32I_LB, RV32I_LH, RV32I_LW, RV32I_LBU,
 RV32I_LHU, RV32I_SB, RV32I_SH, RV32I_SW, RV32I_ADDI,
 RV32I_SLTI, RV32I_SLTIU, RV32I_XORI, RV32I_ORI,
 RV32I_ANDI, RV32I_SLLI, RV32I_SRLI, RV32I_SRAI,RV32I_ADD,
  RV32I_SUB, RV32I_SLL, RV32I_SLT, RV32I_SLTU, RV32I_XOR,
 RV32I_SRL, RV32I_SRA, RV32I_OR, RV32I_AND, RV32I_FENCE,
 RV32I_ECALL, RV32I_EBREAK;
assign RV32I_LUI      = is_rv32_insn && (
 riscv_instr_opcode == 7'b0110111);
assign RV32I_AUIPC    = is_rv32_insn && (
 riscv_instr_opcode == 7'b0010111);
assign RV32I_LB       = is_rv32_insn && (
 riscv_instr_funct3 == 3'b000) & (riscv_instr_opcode == 7'
 b0000011) && (riscv_imm_i_type[1:0] == '0);
assign RV32I_LH       = is_rv32_insn && (
 riscv_instr_funct3 == 3'b001) & (riscv_instr_opcode == 7'
 b0000011) && (riscv_imm_i_type[1:0] == '0);
assign RV32I_LW       = is_rv32_insn && (
 riscv_instr_funct3 == 3'b010) & (riscv_instr_opcode == 7'
 b0000011) && (riscv_imm_i_type[1:0] == '0);
```

```
assign RV32I_LBU      = is_rv32_insn && (
 riscv_instr_funct3 == 3'b100) & (riscv_instr_opcode == 7'
 b0000011) && (riscv_imm_i_type[1:0] == '0);
assign RV32I_LHU      = is_rv32_insn && (
 riscv_instr_funct3 == 3'b101) & (riscv_instr_opcode == 7'
 b0000011) && (riscv_imm_i_type[1:0] == '0);
assign RV32I_SB       = is_rv32_insn && (
 riscv_instr_funct3 == 3'b000) & (riscv_instr_opcode == 7'
 b0100011) && (riscv_imm_s_type[1:0] == '0);
assign RV32I_SH       = is_rv32_insn && (
 riscv_instr_funct3 == 3'b001) & (riscv_instr_opcode == 7'
 b0100011) && (riscv_imm_s_type[1:0] == '0);
assign RV32I_SW       = is_rv32_insn && (
 riscv_instr_funct3 == 3'b010) & (riscv_instr_opcode == 7'
 b0100011) && (riscv_imm_s_type[1:0] == '0);
assign RV32I_ADDI     = is_rv32_insn && (
 riscv_instr_funct3 == 3'b000) & (riscv_instr_opcode == 7'
 b0010011);
assign RV32I_SLTI     = is_rv32_insn && (
 riscv_instr_funct3 == 3'b010) & (riscv_instr_opcode == 7'
 b0010011);
assign RV32I_SLTIU    = is_rv32_insn && (
 riscv_instr_funct3 == 3'b011) & (riscv_instr_opcode == 7'
 b0010011);
assign RV32I_XORI     = is_rv32_insn && (
 riscv_instr_funct3 == 3'b100) & (riscv_instr_opcode == 7'
 b0010011);
assign RV32I_ORI      = is_rv32_insn && (
 riscv_instr_funct3 == 3'b110) & (riscv_instr_opcode == 7'
 b0010011);
assign RV32I_ANDI     = is_rv32_insn && (
 riscv_instr_funct3 == 3'b111) & (riscv_instr_opcode == 7'
 b0010011);
assign RV32I_SLLI     = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0000000) & (riscv_instr_funct3
 == 3'b001) & (riscv_instr_opcode == 7'b0010011) && (!
 riscv_shamt[5]);
assign RV32I_SRLI     = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0000000) & (riscv_instr_funct3
 == 3'b101) & (riscv_instr_opcode == 7'b0010011) && (!
 riscv_shamt[5]);
```

```
assign RV32I_SRAI      = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0100000) & (riscv_instr_funct3
 == 3'b101) & (riscv_instr_opcode == 7'b0010011) && (!
 riscv_shamt[5]);
assign RV32I_ADD       = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0000000) & (riscv_instr_funct3
 == 3'b000) & (riscv_instr_opcode == 7'b0110011);
assign RV32I_SUB       = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0100000) & (riscv_instr_funct3
 == 3'b000) & (riscv_instr_opcode == 7'b0110011);
assign RV32I_SLL       = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0000000) & (riscv_instr_funct3
 == 3'b001) & (riscv_instr_opcode == 7'b0110011);
assign RV32I_SLT       = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0000000) & (riscv_instr_funct3
 == 3'b010) & (riscv_instr_opcode == 7'b0110011);
assign RV32I_SLTU      = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0000000) & (riscv_instr_funct3
 == 3'b011) & (riscv_instr_opcode == 7'b0110011);
assign RV32I_XOR       = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0000000) & (riscv_instr_funct3
 == 3'b100) & (riscv_instr_opcode == 7'b0110011);
assign RV32I_SRL       = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0000000) & (riscv_instr_funct3
 == 3'b101) & (riscv_instr_opcode == 7'b0110011);
assign RV32I_SRA       = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0100000) & (riscv_instr_funct3
 == 3'b101) & (riscv_instr_opcode == 7'b0110011);
assign RV32I_OR        = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0000000) & (riscv_instr_funct3
 == 3'b110) & (riscv_instr_opcode == 7'b0110011);
assign RV32I_AND       = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0000000) & (riscv_instr_funct3
 == 3'b111) & (riscv_instr_opcode == 7'b0110011);
assign RV32I_FENCE     = is_rv32_insn && (
 riscv_instr_funct3 == 3'b000) & (riscv_instr_opcode == 7'
 b0001111);


logic [31:0] riscv_pc_rdata;
always_ff @ (posedge clk_i) begin
```

```
  if (!rst_ni) begin
        riscv_pc_rdata <= 32'h00100080;  //initial PC
      end else if (rvfi_valid) begin
        riscv_pc_rdata <= rvfi_pc_wdata;
      end
end

wire [31:0] rv32im_rs1_rdata, rv32im_rs2_rdata;
assign rv32im_rs1_rdata = u_top.gen_regfile_ff.
 register_file_i.rf_reg_q[rv32im_instr_rs1];
assign rv32im_rs2_rdata = u_top.gen_regfile_ff.
 register_file_i.rf_reg_q[rv32im_instr_rs2];

assign RV32I_JAL       = is_rv32_insn && (
 riscv_instr_opcode == 7'b1101111) && (riscv_imm_j_type
 [1:0] == '0) && (riscv_imm_j_type!='0);

wire [31:0] RV32I_JALR_next_PC = ((riscv_imm_i_type +
 rv32im_rs1_rdata)& (~1));
wire [31:0] RV32I_BEQ_next_PC = (rv32im_rs1_rdata ==
 rv32im_rs2_rdata) ? (riscv_pc_rdata + riscv_imm_b_type) :
  (riscv_pc_rdata + 4);
wire [31:0] RV32I_BNE_next_PC = (rv32im_rs1_rdata !=
 rv32im_rs2_rdata) ? (riscv_pc_rdata + riscv_imm_b_type) :
  (riscv_pc_rdata + 4);
wire [31:0] RV32I_BLT_next_PC = ($signed(rv32im_rs1_rdata)
  < $signed(rv32im_rs2_rdata)) ? (riscv_pc_rdata +
 riscv_imm_b_type) : (riscv_pc_rdata + 4);
wire [31:0] RV32I_BGE_next_PC = ($signed(rv32im_rs1_rdata)
  >= $signed(rv32im_rs2_rdata)) ? (riscv_pc_rdata +
 riscv_imm_b_type) : (riscv_pc_rdata + 4);
wire [31:0] RV32I_BLTU_next_PC = ($unsigned(
 rv32im_rs1_rdata) < $unsigned(rv32im_rs2_rdata))  ? (
 riscv_pc_rdata + riscv_imm_b_type) : (riscv_pc_rdata + 4)
 ;
wire [31:0] RV32I_BGEU_next_PC = ($unsigned(
 rv32im_rs1_rdata) >= $unsigned(rv32im_rs2_rdata)) ? (
 riscv_pc_rdata + riscv_imm_b_type) : (riscv_pc_rdata + 4)
 ;
```

```
assign RV32I_JALR     = is_rv32_insn && (
 riscv_instr_funct3 == 3'b000) & (riscv_instr_opcode == 7'
 b1100111) && (riscv_imm_i_type[1:0] == '0) && (
 riscv_imm_i_type!='0)  && (RV32I_JALR_next_PC[0] == '0);
assign RV32I_BEQ      = is_rv32_insn && (
 riscv_instr_funct3 == 3'b000) & (riscv_instr_opcode == 7'
 b1100011) && (riscv_imm_b_type[1:0] == '0) && (
 riscv_imm_b_type!='0) && (RV32I_BEQ_next_PC[0] == '0);
assign RV32I_BNE      = is_rv32_insn && (
 riscv_instr_funct3 == 3'b001) & (riscv_instr_opcode == 7'
 b1100011) && (riscv_imm_b_type[1:0] == '0) && (
 riscv_imm_b_type!='0) && (RV32I_BNE_next_PC[0] == '0);
assign RV32I_BLT      = is_rv32_insn && (
 riscv_instr_funct3 == 3'b100) & (riscv_instr_opcode == 7'
 b1100011) && (riscv_imm_b_type[1:0] == '0) && (
 riscv_imm_b_type!='0) && (RV32I_BLT_next_PC[0] == '0);
assign RV32I_BGE      = is_rv32_insn && (
 riscv_instr_funct3 == 3'b101) & (riscv_instr_opcode == 7'
 b1100011) && (riscv_imm_b_type[1:0] == '0) && (
 riscv_imm_b_type!='0) && (RV32I_BGE_next_PC[0] == '0);
assign RV32I_BLTU     = is_rv32_insn && (
 riscv_instr_funct3 == 3'b110) & (riscv_instr_opcode == 7'
 b1100011) && (riscv_imm_b_type[1:0] == '0) && (
 riscv_imm_b_type!='0) && (RV32I_BLTU_next_PC[0] == '0);
assign RV32I_BGEU     = is_rv32_insn && (
 riscv_instr_funct3 == 3'b111) & (riscv_instr_opcode == 7'
 b1100011) && (riscv_imm_b_type[1:0] == '0) && (
 riscv_imm_b_type!='0) && (RV32I_BGEU_next_PC[0] == '0);

logic RV32M_MUL, RV32M_MULH, RV32M_MULHSU, RV32M_MULHU,
 RV32M_DIV, RV32M_DIVU, RV32M_REM, RV32M_REMU;
assign RV32M_MUL      = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0000001) & (riscv_instr_funct3
 == 3'b000) & (riscv_instr_opcode == 7'b0110011);
assign RV32M_MULH     = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0000001) & (riscv_instr_funct3
 == 3'b001) & (riscv_instr_opcode == 7'b0110011);
assign RV32M_MULHSU   = is_rv32_insn && (
 riscv_instr_funct7 == 7'b0000001) & (riscv_instr_funct3
 == 3'b010) & (riscv_instr_opcode == 7'b0110011);
```

```
  assign RV32M_MULHU    = is_rv32_insn && (
   riscv_instr_funct7 == 7'b0000001) & (riscv_instr_funct3
   == 3'b011) & (riscv_instr_opcode == 7'b0110011);
  assign RV32M_DIV      = is_rv32_insn && (
   riscv_instr_funct7 == 7'b0000001) & (riscv_instr_funct3
   == 3'b100) & (riscv_instr_opcode == 7'b0110011);
  assign RV32M_DIVU     = is_rv32_insn && (
   riscv_instr_funct7 == 7'b0000001) & (riscv_instr_funct3
   == 3'b101) & (riscv_instr_opcode == 7'b0110011);
  assign RV32M_REM      = is_rv32_insn && (
   riscv_instr_funct7 == 7'b0000001) & (riscv_instr_funct3
   == 3'b110) & (riscv_instr_opcode == 7'b0110011);
  assign RV32M_REMU     = is_rv32_insn && (
   riscv_instr_funct7 == 7'b0000001) & (riscv_instr_funct3
   == 3'b111) & (riscv_instr_opcode == 7'b0110011);

logic is_specified_RV32I, is_specified_RV32M;
  assign is_specified_RV32I = RV32I_LUI||RV32I_AUIPC||
   RV32I_JAL||RV32I_JALR||RV32I_BEQ||RV32I_BNE||RV32I_BLT||
   RV32I_BGE||RV32I_BLTU||RV32I_BGEU||RV32I_LB||RV32I_LH||
   RV32I_LW||RV32I_LBU||RV32I_LHU||RV32I_SB||RV32I_SH||
   RV32I_SW||RV32I_ADDI||RV32I_SLTI||RV32I_SLTIU||RV32I_XORI
   ||RV32I_ORI||RV32I_ANDI||RV32I_SLLI||RV32I_SRLI||
   RV32I_SRAI||RV32I_ADD||RV32I_SUB||RV32I_SLL||RV32I_SLT||
   RV32I_SLTU||RV32I_XOR||RV32I_SRL||RV32I_SRA||RV32I_OR||
   RV32I_AND||RV32I_FENCE;
  assign is_specified_RV32M = RV32M_MUL||RV32M_MULH||
   RV32M_MULHSU||RV32M_MULHU||RV32M_DIV||RV32M_DIVU||
   RV32M_REM||RV32M_REMU;


  logic RV32C_Right_reserved, RVC_Right_ADDI4SPN,
   RVC_Right_LW, RVC_Right_SW, RVC_Right_ADDI, RVC_Right_JAL
   , RVC_Right_LI, RVC_Right_ADDI16SP, RVC_Right_LUI,
   RVC_Right_SRLI, RVC_Right_SRAI, RVC_Right_ANDI,
   RVC_Right_SUB, RVC_Right_XOR, RVC_Right_OR, RVC_Right_AND
   , RVC_Right_J, RVC_Right_BEQZ, RVC_Right_BNEZ,
   RVC_Right_SLLI, RVC_Right_LWSP, RVC_Right_JR,
   RVC_Right_MV, RVC_Right_JALR, RVC_Right_ADD,
   RVC_Right_SWSP;
```

```
assign RV32C_Right_reserved = is_rv32c_insn && (
 riscv_compressed_instrR == 16'b0) || (
 riscv_compressed_instrR[15:10] == 6'b100111 &&
 riscv_compressed_instrR[1:0] == 2'b01);
assign RVC_Right_ADDI4SPN      =  is_rv32c_insn && (
 riscv_compressed_instrR[15:13]  == 3'b000) && (
 riscv_compressed_instrR[1:0]  == 2'b00) && ({22'd0,
 riscv_compressed_instrR[10:7], riscv_compressed_instrR
 [12:11], riscv_compressed_instrR[5],
 riscv_compressed_instrR[6], 2'b00}!='0);
assign RVC_Right_LW                    =  is_rv32c_insn &&
 (riscv_compressed_instrR[15:13]  == 3'b010) && (
 riscv_compressed_instrR[1:0]  == 2'b00);
assign RVC_Right_SW                    =  is_rv32c_insn &&
 (riscv_compressed_instrR[15:13]  == 3'b110) && (
 riscv_compressed_instrR[1:0]  == 2'b00);
assign RVC_Right_ADDI          =  is_rv32c_insn && (
 riscv_compressed_instrR[15:13]  == 3'b000) && (
 riscv_compressed_instrR[1:0]  == 2'b01);
assign RVC_Right_LI                    =  is_rv32c_insn &&
 (riscv_compressed_instrR[15:13]  == 3'b010) && (
 riscv_compressed_instrR[1:0]  == 2'b01);
assign RVC_Right_ADDI16SP      =  is_rv32c_insn && (
 riscv_compressed_instrR[15:13]  == 3'b011) && (
 riscv_compressed_instrR[1:0]  == 2'b01) && (
 riscv_compressed_instrR[11:7] == 5'd2) && ($signed({ {23{
 riscv_compressed_instrR[12]}}, riscv_compressed_instrR
 [4:3], riscv_compressed_instrR[5],
 riscv_compressed_instrR[2], riscv_compressed_instrR[6],
 4'b0})!='0);
assign RVC_Right_LUI           =  is_rv32c_insn && (
 riscv_compressed_instrR[15:13]  == 3'b011) && (
 riscv_compressed_instrR[1:0]  == 2'b01) && (
 riscv_compressed_instrR[11:7] != 5'd2) && ($signed({ {15{
 riscv_compressed_instrR[12]}}, riscv_compressed_instrR
 [6:2], 12'b0})!='0);
assign RVC_Right_SRLI          =  is_rv32c_insn && (
 riscv_compressed_instrR[15:13]  == 3'b100) && (
 riscv_compressed_instrR[11:10] == 2'b00) && (
 riscv_compressed_instrR[1:0]  == 2'b01) && (!
 riscv_compressed_instrR[12]);
```

```
assign RVC_Right_SRAI          =  is_rv32c_insn && (
 riscv_compressed_instrR[15:13]  == 3'b100) && (
 riscv_compressed_instrR[11:10] == 2'b01) && (
 riscv_compressed_instrR[1:0]  == 2'b01) && (!
 riscv_compressed_instrR[12]);
assign RVC_Right_ANDI          =  is_rv32c_insn && (
 riscv_compressed_instrR[15:13]  == 3'b100) && (
 riscv_compressed_instrR[11:10] == 2'b10) && (
 riscv_compressed_instrR[1:0]  == 2'b01);
assign RVC_Right_SUB           =  is_rv32c_insn && (
 riscv_compressed_instrR[15:10] == 6'b100011 ) && (
 riscv_compressed_instrR[6:5] == 2'b00) && (
 riscv_compressed_instrR[1:0]  == 2'b01);
assign RVC_Right_XOR           =  is_rv32c_insn && (
 riscv_compressed_instrR[15:10] == 6'b100011 ) && (
 riscv_compressed_instrR[6:5] == 2'b01) && (
 riscv_compressed_instrR[1:0]  == 2'b01);
assign RVC_Right_OR            =  is_rv32c_insn && (
 riscv_compressed_instrR[15:10] == 6'b100011 ) && (
 riscv_compressed_instrR[6:5] == 2'b10) && (
 riscv_compressed_instrR[1:0]  == 2'b01);
assign RVC_Right_AND           =  is_rv32c_insn && (
 riscv_compressed_instrR[15:10] == 6'b100011 ) && (
 riscv_compressed_instrR[6:5] == 2'b11) && (
 riscv_compressed_instrR[1:0]  == 2'b01);
assign RVC_Right_SLLI          =  is_rv32c_insn && (
 riscv_compressed_instrR[15:13]  == 3'b000 ) && (
 riscv_compressed_instrR[1:0]  == 2'b10) && (!
 riscv_compressed_instrR[12]);
assign RVC_Right_LWSP          =  is_rv32c_insn && (
 riscv_compressed_instrR[15:13]  == 3'b010 ) && (
 riscv_compressed_instrR[1:0]  == 2'b10) && (
 riscv_compressed_instrR[11:7] != '0);
assign RVC_Right_MV                    =  is_rv32c_insn &&
 (riscv_compressed_instrR[15:12] == 4'b1000 ) && (
 riscv_compressed_instrR[1:0]  == 2'b10) && (
 riscv_compressed_instrR[6:2] != '0);
assign RVC_Right_ADD           =  is_rv32c_insn && (
 riscv_compressed_instrR[15:12] == 4'b1001 ) && (
 riscv_compressed_instrR[1:0]  == 2'b10)&& (
 riscv_compressed_instrR[6:2] != '0);
```

```
assign RVC_Right_SWSP =  (riscv_compressed_instrR[15:13]
 == 3'b110 ) && (riscv_compressed_instrR[1:0]  == 2'b10);

// Left 16 bits of a 32-bit RVC instruction
wire [31:0] RVC_Left_JAL_imm = $signed({ {21{
 riscv_compressed_instrL[12]}}, riscv_compressed_instrL
 [8], riscv_compressed_instrL[10], riscv_compressed_instrL
 [9], riscv_compressed_instrL[6], riscv_compressed_instrL
 [7], riscv_compressed_instrL[2], riscv_compressed_instrL
 [11], riscv_compressed_instrL[5], riscv_compressed_instrL
 [4], riscv_compressed_instrL[3], 1'b0});
wire [31:0] RVC_Left_JAL_next_PC = (riscv_pc_rdata +
 RVC_Left_JAL_imm);

wire [31:0] RVC_Left_J_imm = {{21{riscv_compressed_instrL
 [12]}}, riscv_compressed_instrL[8],
 riscv_compressed_instrL[10], riscv_compressed_instrL[9],
 riscv_compressed_instrL[6], riscv_compressed_instrL[7],
 riscv_compressed_instrL[2], riscv_compressed_instrL[11],
 riscv_compressed_instrL[5], riscv_compressed_instrL[4],
 riscv_compressed_instrL[3], 1'b0};
wire [31:0] RVC_Left_J_next_PC = (riscv_pc_rdata +
 RVC_Left_J_imm);

wire [4:0] RVC_Left_B_insn_rs1 = {1'b1,
 riscv_compressed_instrL[9:7]};
wire [31:0] RVC_Left_B_rs1_rdata = u_top.gen_regfile_ff.
 register_file_i.rf_reg_q[RVC_Left_B_insn_rs1];
wire [31:0] RVC_Left_B_imm = { {24{riscv_compressed_instrL
 [12]}}, riscv_compressed_instrL[6:5],
 riscv_compressed_instrL[2], riscv_compressed_instrL
 [11:10], riscv_compressed_instrL[4:3], 1'b0};
wire [31:0] RVC_Left_BEQZ_next_PC = ( (
 RVC_Left_B_rs1_rdata == 0) ? (riscv_pc_rdata +
 RVC_Left_B_imm) : (riscv_pc_rdata + 32'd2) );
wire [31:0] RVC_Left_BNEZ_next_PC = ( (
 RVC_Left_B_rs1_rdata != 0) ? (riscv_pc_rdata +
 RVC_Left_B_imm) : (riscv_pc_rdata + 32'd2) );

wire [4:0] RVC_Left_J_insn_rs1 = riscv_compressed_instrL
 [11:7];
```

```
wire [31:0] RVC_Left_J_rs1_rdata = u_top.gen_regfile_ff.
 register_file_i.rf_reg_q[RVC_Left_J_insn_rs1];
wire [31:0] RVC_Left_JR_next_PC = (RVC_Left_J_rs1_rdata &
 ~32'd1);
wire [31:0] RVC_Left_JALR_next_PC = (RVC_Left_J_rs1_rdata
 & ~1);

assign RVC_Left_JAL           =  is_rv32c_insn && (
 riscv_compressed_instrL[15:13]  == 3'b001) && (
 riscv_compressed_instrL[1:0]  == 2'b01) && (
 RVC_Left_JAL_next_PC[0] == '0);

assign RVC_Left_J                     =  is_rv32c_insn &&
 (riscv_compressed_instrL[15:13]  == 3'b101 ) && (
 riscv_compressed_instrL[1:0]  == 2'b01) && (
 RVC_Left_J_next_PC[0] == '0);

assign RVC_Left_BEQZ          =  is_rv32c_insn && (
 riscv_compressed_instrL[15:13]  == 3'b110 ) && (
 riscv_compressed_instrL[1:0]  == 2'b01) && (
 RVC_Left_BEQZ_next_PC[0] == '0);

assign RVC_Left_BNEZ          =  is_rv32c_insn && (
 riscv_compressed_instrL[15:13]  == 3'b111 ) && (
 riscv_compressed_instrL[1:0]  == 2'b01) && (
 RVC_Left_BNEZ_next_PC[0] == '0);

assign RVC_Left_JR                    =  is_rv32c_insn &&
 (riscv_compressed_instrL[15:12] == 4'b1000 ) && (
 riscv_compressed_instrL[1:0]  == 2'b10) && (
 riscv_compressed_instrL[11:7] != '0) && (
 riscv_compressed_instrL[6:2] == '0) && (
 RVC_Left_JR_next_PC[0] =='0);

assign RVC_Left_JALR          =  is_rv32c_insn && (
 riscv_compressed_instrL[15:12] == 4'b1001 ) && (
 riscv_compressed_instrL[1:0]  == 2'b10) && (
 riscv_compressed_instrL[11:7] != '0) && (
 riscv_compressed_instrL[6:2] == '0) && (
 RVC_Left_JALR_next_PC[0] =='0);
```

```
// Right 16 bits of a 32-bit RVC instruction
wire [31:0] RVC_Right_JAL_imm = $signed({ {21{
 riscv_compressed_instrR[12]}}, riscv_compressed_instrR
 [8], riscv_compressed_instrR[10], riscv_compressed_instrR
 [9], riscv_compressed_instrR[6], riscv_compressed_instrR
 [7], riscv_compressed_instrR[2], riscv_compressed_instrR
 [11], riscv_compressed_instrR[5], riscv_compressed_instrR
 [4], riscv_compressed_instrR[3], 1'b0});
wire [31:0] RVC_Right_JAL_next_PC = (riscv_pc_rdata +
 RVC_Right_JAL_imm);

wire [31:0] RVC_Right_J_imm = {{21{riscv_compressed_instrR
 [12]}}, riscv_compressed_instrR[8],
 riscv_compressed_instrR[10], riscv_compressed_instrR[9],
 riscv_compressed_instrR[6], riscv_compressed_instrR[7],
 riscv_compressed_instrR[2], riscv_compressed_instrR[11],
 riscv_compressed_instrR[5], riscv_compressed_instrR[4],
 riscv_compressed_instrR[3], 1'b0};
wire [31:0] RVC_Right_J_next_PC = (riscv_pc_rdata +
 RVC_Right_J_imm);

wire [4:0] RVC_Right_B_insn_rs1 = {1'b1,
 riscv_compressed_instrR[9:7]};
wire [31:0] RVC_Right_B_rs1_rdata = u_top.gen_regfile_ff.
 register_file_i.rf_reg_q[RVC_Right_B_insn_rs1];
wire [31:0] RVC_Right_B_imm = { {24{
 riscv_compressed_instrR[12]}}, riscv_compressed_instrR
 [6:5], riscv_compressed_instrR[2],
 riscv_compressed_instrR[11:10], riscv_compressed_instrR
 [4:3], 1'b0};
wire [31:0] RVC_Right_BEQZ_next_PC = ( (
 RVC_Right_B_rs1_rdata == 0) ? (riscv_pc_rdata +
 RVC_Right_B_imm) : (riscv_pc_rdata + 32'd2) );
wire [31:0] RVC_Right_BNEZ_next_PC = ( (
 RVC_Right_B_rs1_rdata != 0) ? (riscv_pc_rdata +
 RVC_Right_B_imm) : (riscv_pc_rdata + 32'd2) );

wire [4:0] RVC_Right_J_insn_rs1 = riscv_compressed_instrR
 [11:7];
```

```
wire [31:0] RVC_Right_J_rs1_rdata = u_top.gen_regfile_ff.
 register_file_i.rf_reg_q[RVC_Right_J_insn_rs1];
wire [31:0] RVC_Right_JR_next_PC = (RVC_Right_J_rs1_rdata
 & ~32'd1);
wire [31:0] RVC_Right_JALR_next_PC = (
 RVC_Right_J_rs1_rdata & ~1);


assign RVC_Right_JAL          =  is_rv32c_insn && (
 riscv_compressed_instrR[15:13]  == 3'b001) && (
 riscv_compressed_instrR[1:0]  == 2'b01) && (
 RVC_Right_JAL_next_PC[0] == '0);

assign RVC_Right_J                    =  is_rv32c_insn &&
 (riscv_compressed_instrR[15:13]  == 3'b101 ) && (
 riscv_compressed_instrR[1:0]  == 2'b01) && (
 RVC_Right_J_next_PC[0] == '0);

assign RVC_Right_BEQZ         =  is_rv32c_insn && (
 riscv_compressed_instrR[15:13]  == 3'b110 ) && (
 riscv_compressed_instrR[1:0]  == 2'b01) && (
 RVC_Right_BEQZ_next_PC[0] == '0);

assign RVC_Right_BNEZ         =  is_rv32c_insn && (
 riscv_compressed_instrR[15:13]  == 3'b111 ) && (
 riscv_compressed_instrR[1:0]  == 2'b01) && (
 RVC_Right_BNEZ_next_PC[0] == '0);

assign RVC_Right_JR                   =  is_rv32c_insn &&
 (riscv_compressed_instrR[15:12] == 4'b1000 ) && (
 riscv_compressed_instrR[1:0]  == 2'b10) && (
 riscv_compressed_instrR[11:7] != '0) && (
 riscv_compressed_instrR[6:2] == '0) && (
 RVC_Right_JR_next_PC[0] =='0);

assign RVC_Right_JALR         =  is_rv32c_insn && (
 riscv_compressed_instrR[15:12] == 4'b1001 ) && (
 riscv_compressed_instrR[1:0]  == 2'b10) && (
 riscv_compressed_instrR[11:7] != '0) && (
 riscv_compressed_instrR[6:2] == '0) && (
 RVC_Right_JALR_next_PC[0] =='0);
```

```
logic is_rv32c_insn;
assign is_rv32c_insn = (instr_rdata_i[1:0] != 2'b11);

logic [15:0] riscv_compressed_instrL,
 riscv_compressed_instrR;
assign riscv_compressed_instrL = instr_rdata_i[31:16];
assign riscv_compressed_instrR = instr_rdata_i[15:0];

logic RV32C_Left_reserved, RVC_Left_ADDI4SPN, RVC_Left_LW,
  RVC_Left_SW, RVC_Left_ADDI, RVC_Left_JAL, RVC_Left_LI,
 RVC_Left_ADDI16SP, RVC_Left_LUI, RVC_Left_SRLI,
 RVC_Left_SRAI, RVC_Left_ANDI, RVC_Left_SUB, RVC_Left_XOR,
  RVC_Left_OR, RVC_Left_AND,  RVC_Left_J, RVC_Left_BEQZ,
 RVC_Left_BNEZ, RVC_Left_SLLI, RVC_Left_LWSP, RVC_Left_JR,
  RVC_Left_MV, RVC_Left_JALR, RVC_Left_ADD, RVC_Left_SWSP;
assign RV32C_Left_reserved = is_rv32c_insn && (
 riscv_compressed_instrL == 16'b0) || (
 riscv_compressed_instrL[15:10] == 6'b100111 &&
 riscv_compressed_instrL[1:0] == 2'b01);
assign RVC_Left_ADDI4SPN      =  is_rv32c_insn && (
 riscv_compressed_instrL[15:13]  == 3'b000) && (
 riscv_compressed_instrL[1:0]  == 2'b00) && ({22'd0,
 riscv_compressed_instrL[10:7], riscv_compressed_instrL
 [12:11], riscv_compressed_instrL[5],
 riscv_compressed_instrL[6], 2'b00}!='0);
assign RVC_Left_LW                     =  is_rv32c_insn &&
 (riscv_compressed_instrL[15:13]  == 3'b010) && (
 riscv_compressed_instrL[1:0]  == 2'b00);
assign RVC_Left_SW                     =  is_rv32c_insn &&
 (riscv_compressed_instrL[15:13]  == 3'b110) && (
 riscv_compressed_instrL[1:0]  == 2'b00);
assign RVC_Left_ADDI          =  is_rv32c_insn && (
 riscv_compressed_instrL[15:13]  == 3'b000) && (
 riscv_compressed_instrL[1:0]  == 2'b01);
assign RVC_Left_LI                     =  is_rv32c_insn &&
 (riscv_compressed_instrL[15:13]  == 3'b010) && (
 riscv_compressed_instrL[1:0]  == 2'b01);
```

```
assign RVC_Left_ADDI16SP      =  is_rv32c_insn && (
 riscv_compressed_instrL[15:13]  == 3'b011) && (
 riscv_compressed_instrL[1:0]  == 2'b01) && (
 riscv_compressed_instrL[11:7] == 5'd2) && ($signed({ {23{
 riscv_compressed_instrL[12]}}, riscv_compressed_instrL
 [4:3], riscv_compressed_instrL[5],
 riscv_compressed_instrL[2], riscv_compressed_instrL[6],
 4'b0})!='0);
assign RVC_Left_LUI           =  is_rv32c_insn && (
 riscv_compressed_instrL[15:13]  == 3'b011) && (
 riscv_compressed_instrL[1:0]  == 2'b01) && (
 riscv_compressed_instrL[11:7] != 5'd2) && ($signed({ {15{
 riscv_compressed_instrL[12]}}, riscv_compressed_instrL
 [6:2], 12'b0})!='0);
assign RVC_Left_SRLI          =  is_rv32c_insn && (
 riscv_compressed_instrL[15:13]  == 3'b100) && (
 riscv_compressed_instrL[11:10] == 2'b00) && (
 riscv_compressed_instrL[1:0]  == 2'b01) && (!
 riscv_compressed_instrL[12]);
assign RVC_Left_SRAI          =  is_rv32c_insn && (
 riscv_compressed_instrL[15:13]  == 3'b100) && (
 riscv_compressed_instrL[11:10] == 2'b01) && (
 riscv_compressed_instrL[1:0]  == 2'b01) && (!
 riscv_compressed_instrL[12]);
assign RVC_Left_ANDI          =  is_rv32c_insn && (
 riscv_compressed_instrL[15:13]  == 3'b100) && (
 riscv_compressed_instrL[11:10] == 2'b10) && (
 riscv_compressed_instrL[1:0]  == 2'b01);
assign RVC_Left_SUB           =  is_rv32c_insn && (
 riscv_compressed_instrL[15:10] == 6'b100011 ) && (
 riscv_compressed_instrL[6:5] == 2'b00) && (
 riscv_compressed_instrL[1:0]  == 2'b01);
assign RVC_Left_XOR           =  is_rv32c_insn && (
 riscv_compressed_instrL[15:10] == 6'b100011 ) && (
 riscv_compressed_instrL[6:5] == 2'b01) && (
 riscv_compressed_instrL[1:0]  == 2'b01);
assign RVC_Left_OR            =  is_rv32c_insn && (
 riscv_compressed_instrL[15:10] == 6'b100011 ) && (
 riscv_compressed_instrL[6:5] == 2'b10) && (
 riscv_compressed_instrL[1:0]  == 2'b01);
```

```
assign RVC_Left_AND           =  is_rv32c_insn && (
 riscv_compressed_instrL[15:10] == 6'b100011 ) && (
 riscv_compressed_instrL[6:5] == 2'b11) && (
 riscv_compressed_instrL[1:0]  == 2'b01);
assign RVC_Left_SLLI          =  is_rv32c_insn && (
 riscv_compressed_instrL[15:13]  == 3'b000 ) && (
 riscv_compressed_instrL[1:0]  == 2'b10) && (!
 riscv_compressed_instrL[12]);
assign RVC_Left_LWSP          =  is_rv32c_insn && (
 riscv_compressed_instrL[15:13]  == 3'b010 ) && (
 riscv_compressed_instrL[1:0]  == 2'b10) && (
 riscv_compressed_instrL[11:7] != '0);
assign RVC_Left_MV                    =  is_rv32c_insn &&
 (riscv_compressed_instrL[15:12] == 4'b1000 ) && (
 riscv_compressed_instrL[1:0]  == 2'b10) && (
 riscv_compressed_instrL[6:2] != '0);
assign RVC_Left_ADD           =  is_rv32c_insn && (
 riscv_compressed_instrL[15:12] == 4'b1001 ) && (
 riscv_compressed_instrL[1:0]  == 2'b10)&& (
 riscv_compressed_instrL[6:2] != '0);
assign RVC_Left_SWSP  =  (riscv_compressed_instrL[15:13]
 == 3'b110 ) && (riscv_compressed_instrL[1:0]  == 2'b10);

logic is_specified_RV32C;
assign is_specified_RV32C = (RVC_Left_ADDI4SPN||
 RVC_Left_LW||RVC_Left_SW||RVC_Left_ADDI||RVC_Left_JAL||
 RVC_Left_LI||RVC_Left_ADDI16SP||RVC_Left_LUI||
 RVC_Left_SRLI||RVC_Left_SRAI||RVC_Left_ANDI||RVC_Left_SUB
 ||RVC_Left_XOR||RVC_Left_OR||RVC_Left_AND||RVC_Left_J||
 RVC_Left_BEQZ||RVC_Left_BNEZ||RVC_Left_SLLI||
 RVC_Left_LWSP||RVC_Left_JR||RVC_Left_MV||RVC_Left_JALR||
 RVC_Left_ADD||RVC_Left_SWSP) && (RVC_Right_ADDI4SPN||
 RVC_Right_LW||RVC_Right_SW||RVC_Right_ADDI||RVC_Right_JAL
 ||RVC_Right_LI||RVC_Right_ADDI16SP||RVC_Right_LUI||
 RVC_Right_SRLI||RVC_Right_SRAI||RVC_Right_ANDI||
 RVC_Right_SUB||RVC_Right_XOR||RVC_Right_OR||RVC_Right_AND
 ||RVC_Right_J||RVC_Right_BEQZ||RVC_Right_BNEZ||
 RVC_Right_SLLI||RVC_Right_LWSP||RVC_Right_JR||
 RVC_Right_MV||RVC_Right_JALR||RVC_Right_ADD||
 RVC_Right_SWSP);
```

Appendix G

Static and Dynamic Slicing

Dynamic slicing is an improved method of simulation-based fault injection. Volk uses dynamic slicing to speed up fault injection [54], which has been referenced in Chapter 2. In that paper, Volk claims that with dynamic slicing, up to 10 percent fault list reduction can be achieved, which is a significant improvement in validation costs. Such a methodology is adopted in this chapter to prove its limitations.

The adopted methodology is different from Volk's work. We used Verilator to simulate the design and to collect coverage data. In addition, we developed a Python program to process the coverage data to automatically generate static slices and dynamic slices. The commercial application functions (except simulation) used in Volk's work are replaced by the developed Python program.

The adopted methodology can be divided into five steps:

1. Build corresponding static slices for all primary core outputs.
2. Run simulations to collect coverage data at each clock cycle. The coverage data contains the lines of consumed/executed code at each clock cycle.
3. Analyse the coverage data to identify the consumed registers (the registers in the lines of consumed code) at each clock cycle.
4. Generate dynamic slices by comparing the static slices with the consumed registers.
5. Prune the fault list with the help of the dynamic slices and perform simulation-based fault injection with the pruned fault list.

The first step is to generate a static slice for each sink. Since we are interested in monitoring the response of faults, a sink in this experiment is an output port of the Ibex Core. A static slice is a collection of all source code that may influence the sink

[54]. In this experiment, we are interested in SEUs, hence a static slice is narrowed down to a collection of all registers that may influence the chosen output port, regardless of the benchmark. A static slice is similar to a COI introduced in Chapter 2. A script is used to automatically generate static slices. The Breadth-First Search algorithm is implemented in the script to generate static slices of all sinks, as shown in the following:

a) Traverse the entire source code to find all (first-level) registers that can influence the target sink. Append the (first-level) registers to a list.

b) For each register in the list, traverse the entire source code to find all (second-level) registers that can influence the target register. Append the (second-level) registers to the list.

c) Repeat step b) until no more sub-level registers can be found and all the registers in the list have been processed. The list is a static slice for the target sink.

d) Go back to step a) to generate static slices of the other sinks.

The second step is to collect coverage data. CoreMark is chosen as the benchmark. Verilator is used to simulate the Ibex Core and collect coverage data. Verilator supports block coverage, line coverage, and branch coverage. In this experiment, only line coverage is used, which records lines of code that are executed/consumed during simulation. Consumed registers can be easily extracted from the consumed code lines by searching register names in the corresponding source code. An SEU can occur at any clock cycle during simulation. In order to prune the fault list, it is necessary to list the consumed registers in each clock cycle. As a result, line coverage data at each clock cycle are recorded. Collecting coverage data in this step relies on Verilator. However, Verilator has limited support to generate coverage data for each bit: it cannot identify more than two-level depths if SystemVerilog structures are used to declare registers. For example, the Ibex Core uses a packaged structure to declare a CPU control register, as shown in the following.

```
typedef struct packed {
  logic [2:0]  dummy_instr_mask;
  logic        dummy_instr_en;
  logic        data_ind_timing;
  logic        icache_enable;
} cpu_ctrl_t;
cpu_ctrl_t   cpuctrl_q;
```

`cpuctrl_q` is a register with two-level depths. The structure `cpu_ctrl_t` defines all fields of the register. Verilator is able to generate the coverage data of all the fields of this register. However, if one of the field is another structure, this register is more than two-level depths and Verilator cannot trace the nested structure. As a result, there is a limitation in this step: it is hard to narrow consumed registers down to consumed bits.

The third step is to analyse the coverage data to identify consumed registers at each clock cycle. The coverage data generated in the second step contains consumed line numbers at each clock cycle with corresponding source file names. Actual executed/consumed codes are not included. In other words, consumed registers cannot be directly extracted from the line coverage data. Hence, a script was developed to process the coverage data. For each coverage data at each clock cycle, the script first extracts the executed line numbers and file names from the coverage data. Then corresponding consumed codes in the source files are located. The next step is to identify and record the consumed registers from the consumed codes. The outputs of this step are multiple lists of consumed registers at each clock cycle for the given benchmark and different sinks.

There are many lines of source code in a complex design. For a certain program/benchmark, not all lines of code are executed during simulation. For instance, not all branches are executed because of some unmet conditions. Similarly, though there may exist many registers in a static slice, not all registers are actually consumed during simulation. A dynamic slice is a collection of actually executed lines for a benchmark [54]. For convenience in this Chapter, a dynamic slice is narrowed down to a collection of actually consumed registers during simulation for a benchmark. Because we are interested in SEUs in registers inside the Ibex Core; other components in the executed lines are out of consideration.

The fourth step is to compare the static slice with the list of consumed registers generated from the third step. The static slice is a list of registers that may influence the chosen output. By comparing the static slice and the list of consumed registers, consumed registers that actually influence the sink at each clock cycle can be found. Such a list of registers is a dynamic slice. For a chosen sink and a benchmark, there is one static slice and multiple dynamic slices, because there are multiple clock cycles in simulation. The number of dynamic slices depends on the clock cycles of the benchmark.

The next step is to generate a fault list from the dynamic slices. Each dynamic slice contains all registers that influence an output port at a certain clock cycle, hence fault locations and injection time can be specified. Faults outside the dynamic slices are pruned from the fault list. The output of this step is a fault list that contains both fault location and injection time for the chosen benchmark. The fault list is then used for simulation-based fault injection.

There are nine output ports of the Ibex Core. It only takes seconds to generate all static slices. However, it takes about 16 seconds to write all coverage data for one clock cycle to the hard disk. According to this calculation, it may take up to 777 days to record all coverage data, which is unacceptable. To reduce time, the multiplication program from Mibench benchmark is used instead. Simulating the multiplication program takes 68490 clock cycles, which is far less than CoreMark. Each coverage data takes around 195.1 KB. As a result, it takes approximately 13 days and 13 GB to record all coverage data. Then the above steps are performed. The following is an example for demonstrating the methodology. To save space, only a part of the results are shown.

```
==========Static Slice Example=====================
Sink: instr_req_o

Module: ibex_core
register: instr_req_o
Parents in current module:
instr_req_out pmp_req_err

Module: ibex_prefetch_buffer
Parents_in_current_module:
valid_req_q rdata_outstanding_q rdata_pmp_err_q

Module: ibex_fetch_fifo
Parents_in_current_module:
valid_q out_addr_o instr_addr_q rdata_q err_q

===================================================

==========Coverage Data Example====================
Bit Name            Clock Cycle
ctrl_fsm_cs[0]      1
counter_q[0]        1
counter_q[0]        1
mtvec_q[0]          1
mtvec_q[20]         1
rdata_q[0]          1
rdata_q[20]         1
instr_addr_q[20]    1
instr_addr_q[7]     1
fetch_addr_q[20]    1
fetch_addr_q[7]     1
ctrl_fsm_cs[0]      2
dcsr_q.prv              2
===================================================
==========Dynamic Slice Example====================
register     cycle module
rdata_q      1     ibex_fetch_fifo
instr_addr_q 1     ibex_fetch_fifo
rdata_q      2     ibex_fetch_fifo
==================================================
```

The primary output `instr_req_o`, which is an instruction request signal to the memory, is chosen as the sink. The first step is to generate a static slice. The script firstly searched all (first-level) registers that may affect the sink and found registers `instr_req_out` and `pmp_req_err` in the `ibex_core` module. Then it searched registers that may affect the (first-level) registers and found multiple (second-level) registers in the `ibex_prefetch_buffer` module and (third-level) registers in the `ibex_fetch_fifo` module. The coverage data is supposed to record all consumed 'bits' at each clock cycle. However, as noted above, Verilator cannot trace bits in nested structures, such as `dcsr_q.prv`. `dcsr_q` is a CSR register used in Debug mode. The Ibex Core uses nested SystemVerilog structures to declare this register. `prv` is a two-bit register that stores the privilege mode. It is hard to determine which bit is the real consumed bit from the coverage data. Hence, we only recorded registers in the dynamic slice example.

In theory, only consumed registers in each cycle should affect the sink. By removing unconsumed registers from the whole register list at each cycle, the fault list should be pruned. According to the result, there are 33 to 66 consumed registers in each clock cycle. The variation is caused by different sizes of static slices of different outputs. In general, most registers are consumed at each clock cycle. After 68490 clock cycles, there are 2324327 consumed registers. The register reduction percentage is 52 %.

The reduction percentage is different from [54]. Note this is the register reduction percentage for a simple multiplication program instead of the bit reduction percentage for all benchmarks. There are 1,435,387 faults left in the fault list after performing dynamic slicing. Compared to the total 137,527,920 faults without dynamic slicing, the reduction is significant. The fault list is reduced by almost 99 %. The reduction percentage is high because a simple program is simulated. Using a simple multiplication program saves time and space. However, there are registers that are never consumed. For example, register `instr_is_compressed_id_o`, which contains compressed instructions. This register is never consumed because after compilation there are no compressed instructions in the simple multiplication program. As a result, further fault injection has limited fault coverage.

Using dynamic slicing does prune the fault list. However, it is still practically impossible to simulate the remaining 1,435,387 faults in the fault list. The cost of reducing the fault list is overhead time and storage space. It may be helpful to use dynamic slicing to find non-critical faults for a design running a certain program. However, fault injection is still needed to test the rest faults, since the results of the rest faults are unknown. It is not worth using dynamic slicing to prune the fault list, because after the efforts of pruning, it is still practically impossible to use fault injection to test the remaining faults.

Dynamic slices generated above are just for one multiplication program. By comparing the static slices and dynamic slices mentioned above, most lines of codes (and hence registers) are consumed. However, there are some registers that are never consumed, leading to unreachable states or corner cases. To reach corner cases or to cover all state space, careful stimulus and hence more programs, dynamic slicing, and more fault injections are needed, leading to infeasible time and storage space overhead.

In summary, applying dynamic slicing can prune the fault list. However, this method relies on collecting coverage data. It is hard to apply this method with tools that cannot support the bit coverage well, such as Verilator in this experiment. The prune percentage varies based on designs and programs. In general, it is not worth using dynamic slicing to prune the fault list for further fault injection. In addition, this method does not solve any of the limitations of simulation-based fault injection. It is not appropriate to use this improved simulation-based fault injection method to achieve the objectives of this research.

Appendix H

Source code of extra MULTDIV

```
//Extra MULTIDV without MAC
logic [63:0] mul_result_ext;
logic [31:0] mul_result;
logic [31:0] Q, R;
logic [ResidueWidth-1:0] Q_residue, R_residue;
logic [1:0] a_residue, b_residue;
logic [3:0] residue_mul;

always_comb begin
mul_result_ext = op_a_i[31:0]*op_b_i[31:0];
mul_result = mul_result_ext[63:32];
Q = $signed(op_a_i[31:0]) / $signed(op_b_i[31:0]);
R = $signed(op_a_i[31:0]) % $signed(op_b_i[31:0]);
Q_residue = Q%3;
R_residue = R%3;
a_residue = op_a_i[33:32];
b_residue = op_b_i[33:32];
residue_mul = a_residue*b_residue;
multdiv_result_o = residue_mul%3;
```

References

[1] A.V. Jayakumar and C. Elks. Property-based fault injection: A novel approach to model-based fault injection for safety critical systems. In Marc Zeller and Kai Höfig, editors, Model-Based Safety and Assessment, pages 115–129, Cham, 2020. Springer International Publishing. ISBN 978-3-030-58920-2.

[2] S. Marchese, J. Grosse, and OneSpin. Formal fault propagation analysis that scales to modern automotive socs. 2017. URL https://api.semanticscholar.org/CorpusID:250647228.

[3] A.Q.Dao, M.P.H. Lin, and A. Mishchenko. Sat-based fault equivalence checking in functional safety verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37:3198–3205, 2018. URL https://api.semanticscholar.org/CorpusID:53781579.

[4] W. Hu, J. Tan, L. Wu, Y. Tai, and L. Hong. Developing formal models for measuring fault effects using functional eda tools. In 2021 IEEE International Test Conference in Asia (ITC-Asia), pages 1–6, 2021. doi: 10.1109/ITC-Asia53059.2021.9808799.

[5] E. Kaja, N. Gerlin, B. Zhao, D.S. Lopera, J.A. Halabi, A.S. Khan, S. Prebeck, D. Stoffel, W. Kunz, and W. Ecker. An automated exhaustive fault analysis technique guided by processor formal verification methods. In 2024 25th International Symposium on Quality Electronic Design (ISQED), pages 1–8, 2024. doi: 10.1109/ISQED60706.2024.10528697.

[6] RISC-V Instruction Set Manual, Volume I: User-Level ISA, .

[7] B. Xue and M. Zwolinski. Using formal methods to evaluate hardware reliability in the presence of soft errors. In 2022 17th Conference on Ph.D Research in Microelectronics and Electronics (PRIME), pages 29–32, 2022. doi: 10.1109/PRIME55000.2022.9816775.

[8] S. Mittal and J. S. Vetter. A Survey of Techniques for Modeling and Improving Reliability of Computing Systems. IEEE Transactions on Parallel and Distributed Systems, 27(4):1226–1238, April 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2015.2426179.

[9] R. Drechsler. PolyAdd: Polynomial Formal Verification of Adder Circuits. In 2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), pages 99–104, 2021. doi: 10.1109/DDECS52668.2021.9417052.

[10] A. Dixit and A. Wood. The impact of new technology on soft error rates. In 2011 International Reliability Physics Symposium, pages 5B.4.1–5B.4.7, April 2011. doi: 10.1109/IRPS.2011.5784522.

[11] A. Vijayan, S. Kiamehr, M. Ebrahimi, K. Chakrabarty, and M. B. Tahoori. Online Soft-Error Vulnerability Estimation for Memory Arrays and Logic Cores. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37 (2):499–511, Feb 2018. ISSN 0278-0070.

[12] X. Iturbe, B. Venu, and E. Ozer. Soft error vulnerability assessment of the real-time safety-related ARM Cortex-R5 CPU. In 2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pages 91–96, Sep. 2016. doi: 10.1109/DFT.2016.7684076.

[13] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. IEEE Transactions on Device and Materials Reliability, 5(3): 305–316, Sep. 2005. ISSN 1530-4388. doi: 10.1109/TDMR.2005.853449.

[14] R. Travessini, P. R. C. Villa, F. L. Vargas, and E. A. Bezerra. Processor core profiling for SEU effect analysis. In 2018 IEEE 19th Latin-American Test Symposium (LATS), pages 1–6, March 2018.

[15] IEEE Standard Classification for Software Anomalies. IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), pages 1–23, 2010. doi: 10.1109/IEEESTD.2010.5399061.

[16] A. Bernardini, W. Ecker, and U. Schlichtmann. Where formal verification can help in functional safety analysis. In 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pages 1–8, 2016. doi: 10.1145/2966986.2980087.

[17] E. Touloupis, J. A. Flint, Member, V. A. Chouliaras, and D. D. Ward. Study of the Effects of SEU-Induced Faults on a Pipeline Protected Microprocessor. IEEE Transactions on Computers, 56(12):1585–1596, Dec 2007. ISSN 0018-9340. doi: 10.1109/TC.2007.70766.

[18] G.S. Rodrigues, F.L. Kastensmidt, and A. Bosio. Radiation Effects on Digital Devices, pages 25–36. Springer International Publishing, Cham, 2022. ISBN 978-3-031-15717-2. doi: 10.1007/978-3-031-15717-2_3. URL `https://doi.org/10.1007/978-3-031-15717-2_3`.

[19] M. Dumont, M. Lisart, and P. Maurine. Electromagnetic Fault Injection : How Faults Occur. In 2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pages 9–16, 2019. doi: 10.1109/FDTC.2019.00010.

[20] H. Zhuang, R. Bauer, and M. Dinkel. Electromigration in Power Devices: A Combined Effect of Electromigration and Thermal Migration. Journal of Microelectronics and Electronic Packaging, 18(1):1–6, 04 2021. ISSN 1551-4897. doi: 10.4071/imaps.1377365. URL `https://doi.org/10.4071/imaps.1377365`.

[21] D.G. Pierce and P.G. Brusius. Electromigration: A review. Microelectronics Reliability, 37(7):1053–1072, 1997. ISSN 0026-2714. doi: https://doi.org/10.1016/S0026-2714(96)00268-5. URL `https://www.sciencedirect.com/science/article/pii/S0026271496002685`. Reliability Physics of Advanced Electron Devices.

[22] K. Tu. Electromigration-induced failure in Al and Cu interconnects, page 270–288. Cambridge University Press, 2010. doi: 10.1017/CBO9780511777691.013.

[23] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing, 1(1):11–33, Jan 2004. ISSN 1545-5971. doi: 10.1109/TDSC.2004.2.

[24] H. Ziade, R. Ayoubi, and R. Velazco. A Survey on Fault Injection Techniques. The International Arab Journal of Information Technology, 1:171–186, 01 2004.

[25] P.D. Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), pages 1–8, 2017. doi: 10.1109/PATMOS.2017.8106976.

[26] lowRISC. Ibex: An embedded 32 bit RISC-V CPU core. URL `https://github.com/lowRISC/ibex`.

[27] A. Mukati. A survey of memory error correcting techniques for improved reliability. Journal of Network and Computer Applications, 34(2):517–522, 2011. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2010.11.006. URL `https://www.sciencedirect.com/science/article/pii/S1084804510002043`. Efficient and Robust Security and Services of Wireless Mesh Networks.

[28] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In 2019 IEEE Symposium on Security and Privacy (SP), pages 55–71. IEEE, 2019.

[29] S.G. Amrutha and V.S. Chakravarthi. Design of error correction engine based on flexible unequal error control code (fuec) for flash memory faults in space applications. In Shubhakar Kalya, Muralidhar Kulkarni, and K.S. Shivaprakasha, editors, Advances in Communication, Signal Processing, VLSI, and Embedded Systems, pages 419–431, Singapore, 2020. Springer Singapore. ISBN 978-981-15-0626-0.

[30] D.L. Perry and H. Foster. Applied Formal Verification: For Digital Circuit Design. McGraw-Hill electronic engineering series. McGraw-Hill Education, 2005. ISBN 9780071588898. URL https://books.google.co.uk/books?id=imKnsuYmZMkC.

[31] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. SWIFT: software implemented fault tolerance. In International Symposium on Code Generation and Optimization, pages 243–254, 2005. doi: 10.1109/CGO.2005.34.

[32] R. Jeyapaul, R. Flores, A. Avila, and A. Shrivastava. Systematic Methodology for the Quantitative Analysis of Pipeline-Register Reliability. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 25(2):547–555, Feb 2017. ISSN 1063-8210. doi: 10.1109/TVLSI.2016.2574642.

[33] C. Bottoni, B. Coeffic, J. Daveau, L. Naviner, and P. Roche. Partial triplication of a SPARC-V8 microprocessor using fault injection. In 2015 IEEE 6th Latin American Symposium on Circuits Systems (LASCAS), pages 1–4, Feb 2015. doi: 10.1109/LASCAS.2015.7250415.

[34] ISO 26262-1: Road vehicles — Functional safety.

[35] T. Bonnoit, A. Coelho, N. Zergainoh, and R. Velazco. SEU impact in processor's control-unit: Preliminary results obtained for LEON3 soft-core. In 2017 18th IEEE Latin American Test Symposium (LATS), pages 1–4, March 2017. doi: 10.1109/LATW.2017.7906763.

[36] H. Cho. Impact of Microarchitectural Differences of RISC-V Processor Cores on Soft Error Effects. IEEE Access, 6:41302–41313, 2018. ISSN 2169-3536.

[37] M. Rebaudengo, M.S. Reorda, and M. Violante. Accurate Analysis of Single Event Upsets in a Pipelined Microprocessor. Journal of Electronic Testing, 19(5): 577–584, Oct 2003. ISSN 1573-0727. doi: 10.1023/A:1025130131636. URL https://doi.org/10.1023/A:1025130131636.

[38] S. Satoh, Y. Tosaka, and S.A. Wender. Geometric effect of multiple-bit soft errors induced by cosmic ray neutrons on DRAM's. IEEE Electron Device Letters, 21 (6):310–312, 2000. doi: 10.1109/55.843160.

[39] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong. Characterization of multi-bit soft error events in advanced SRAMs. In IEEE International Electron Devices Meeting 2003, pages 21.4.1–21.4.4, 2003. doi: 10.1109/IEDM.2003.1269335.

[40] R. Koga, S.H. Penzin, K.B. Crawford, and W.R. Crain. Single event functional interrupt (SEFI) sensitivity in microcircuits. In RADECS 97. Fourth European Conference on Radiation and its Effects on Components and Systems (Cat. No.97TH8294), pages 311–318, 1997. doi: 10.1109/RADECS.1997.698915.

[41] J. Benedetto, P. Eaton, K. Avery, D. Mavis, M. Gadlage, T. Turflinger, P.E. Dodd, and G. Vizkelethyd. Heavy ion-induced digital single-event transients in deep submicron processes. IEEE Transactions on Nuclear Science, 51(6):3480–3485, 2004. doi: 10.1109/TNS.2004.839173.

[42] P.E. Dodd, M.R. Shaneyfelt, J.R. Schwank, and G.L. Hash. Neutron-induced latchup in SRAMs at ground level. In 2003 IEEE International Reliability Physics Symposium Proceedings, 2003. 41st Annual., pages 51–55, 2003. doi: 10.1109/RELPHY.2003.1197720.

[43] A. Ramos, J.A. Maestro, and P. Reviriego. Characterizing a RISC-V SRAM-based FPGA implementation against Single Event Upsets using fault injection. Microelectronics Reliability, 78:205 – 211, 2017. ISSN 0026-2714.

[44] B. Sangchoolie, K. Pattabiraman, and J. Karlsson. One Bit is (Not) Enough: An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors. In 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 97–108, June 2017.

[45] Z. Navabi. Fault Simulation Applications and Methods, pages 103–142. Springer US, Boston, MA, 2011. ISBN 978-1-4419-7548-5. doi: 10.1007/978-1-4419-7548-5_4. URL https://doi.org/10.1007/978-1-4419-7548-5_4.

[46] H.K. Lee and D.S. Ha. Hope: an efficient parallel fault simulator for synchronous sequential circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 15(9):1048–1058, 1996. doi: 10.1109/43.536711.

[47] U. Reinsalu, J. Raik, R. Ubar, and P. Ellervee. Fast RTL Fault Simulation Using Decision Diagrams and Bitwise Set Operations. In 2011 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, pages 164–170, 2011. doi: 10.1109/DFT.2011.42.

[48] J. Seaton. Fault simulation basics. In Proceedings., Second Annual IEEE ASIC Seminar and Exhibit,, pages T5–1/1, 1989. doi: 10.1109/ASIC.1989.123161.

[49] Y. Kilic and M. Zwolinski. Behavioral fault modeling and simulation using VHDL-AMS to speed-up analog fault simulation. Analog Integrated Circuits and Signal Processing, 39(2):177–190, May 2004. URL `https://eprints.soton.ac.uk/259464/`.

[50] V.R. Devanathan, L. Balasubramanian, and R. Parekhji. New Methods for Simulation Speed-up and Test Qualification with Analog Fault Simulation. In 2015 28th International Conference on VLSI Design, pages 363–368, 2015. doi: 10.1109/VLSID.2015.67.

[51] O. Sinanoglu and A. Orailoglu. RT-level fault simulation based on symbolic propagation. Proceedings 19th IEEE VLSI Test Symposium. VTS 2001, pages 240–245, 2001. URL `https://api.semanticscholar.org/CorpusID:14356267`.

[52] M. Karami, M. Haghbayan, M. Ebrahimi, A. Miele, and J. Plosila. Thread-level Parallelism in Fault Simulation of Deep Neural Networks on Multi-Processor Systems. In 2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pages 1–4, 2022. doi: 10.1109/DFT56152.2022.9962358.

[53] J. Laurent, V. Beroulle, C. Deleuze, and F. Pebay-Peyroula. Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures. In 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pages 252–255, March 2019. doi: 10.23919/DATE.2019.8715158.

[54] M. Volk, S. Junges, and J. Katoen. Fast Dynamic Fault Tree Analysis by Model Checking Techniques. IEEE Transactions on Industrial Informatics, 14(1): 370–379, Jan 2018. ISSN 1551-3203. doi: 10.1109/TII.2017.2710316.

[55] E.M. Clarke, T.A. Henzinger, H. Veith, and R. Bloem. Handbook of model checking, volume 10. Springer, 2018.

[56] M. Kooli and G. Di Natale. A survey on simulation-based fault injection tools for complex systems. In 2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS), pages 1–6, May 2014. doi: 10.1109/DTIS.2014.6850649.

[57] R. Natella, D. Cotroneo, and H.S. Madeira. Assessing Dependability with Software Fault Injection: A Survey. ACM Comput. Surv., 48(3):44:1–44:55, February 2016. ISSN 0360-0300. doi: 10.1145/2841425. URL `http://doi.acm.org/10.1145/2841425`.

[58] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos, and P. Rech. Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments. In International Conference on Dependable Systems and Networks (DSN), 06 2019.

[59] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538), pages 3–14, Dec 2001. doi: 10.1109/WWC.2001.990739.

[60] INC. XILINX. Soft Error Mitigation (SEM) Core, 2018. URL `https://www.xilinx.com/products/intellectual-property/sem.html#documentation`.

[61] J. Plusquellic, D.E. Owen, T.J. Mannos, and B. Dziki. Information Leakage Analysis Using a Co-Design-Based Fault Injection Technique on a RISC-V Microprocessor. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 41(3):438–451, 2022. doi: 10.1109/TCAD.2021.3065915.

[62] D.E. Owen, J. Joseph, J. Plusquellic, T.J. Mannos, and B. Dziki. Node Monitoring as a Fault Detection Countermeasure against Information Leakage within a RISC-V Microprocessor. Cryptography, 6(3), 2022. ISSN 2410-387X. doi: 10.3390/cryptography6030038. URL `https://www.mdpi.com/2410-387X/6/3/38`.

[63] C. Kern and M.R. Greenstreet. Formal Verification in Hardware Design: A Survey. ACM Trans. Des. Autom. Electron. Syst., 4(2):123–193, April 1999. ISSN 1084-4309.

[64] T. Kropf. Introduction to Formal Hardware Verification. Springer Berlin Heidelberg, 2013. ISBN 9783662038093. URL `https://books.google.co.uk/books?id=7ImrCAAAQBAJ`.

[65] E.M. Clarke, O. Grumberg, and D.A. Peled. Model Checking. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-03270-8.

[66] M.H. Zaki, S. Tahar, and G. Bois. Formal verification of analog and mixed signal designs: A survey. Microelectronics Journal, 39(12):1395 – 1404, 2008. ISSN 0026-2692. doi: https://doi.org/10.1016/j.mejo.2008.05.013. URL `http://www.sciencedirect.com/science/article/pii/S0026269208002085`.

[67] C. Baier and J. Katoen. Principles of Model Checking (Representation and Mind Series). The MIT Press, 2008. ISBN 026202649X.

[68] S.A. Kripke. Semantical considerations on modal logic. 2012. URL `https://api.semanticscholar.org/CorpusID:56539813`.

[69] R.P. Kurshan. Computer-aided verification of coordinating processes: the automata-theoretic approach. Princeton University Press, USA, 1994. ISBN 0691034362.

[70] F. Cassez, C. Jard, B. Rozoy, and .D. Ryan. Modeling and verification of parallel processes. In Lecture Notes in Computer Science, 2001. URL https://api.semanticscholar.org/CorpusID:33442020.

[71] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 states and beyond. Information and Computation, 98(2): 142–170, 1992. ISSN 0890-5401. doi: https://doi.org/10.1016/0890-5401(92)90017-A. URL https://www.sciencedirect.com/science/article/pii/089054019290017A.

[72] Bryant. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, C-35(8):677–691, 1986. doi: 10.1109/TC.1986.1676819.

[73] A.J. Hu. Formal hardware verification with bdds: an introduction. In 1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, PACRIM. 10 Years Networking the Pacific Rim, 1987-1997, volume 2, pages 677–682 vol.2, 1997. doi: 10.1109/PACRIM.1997.620351.

[74] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In International Conference on Tools and Algorithms for Construction and Analysis of Systems, 1999. URL https://api.semanticscholar.org/CorpusID:524729.

[75] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. IOS Press, NLD, 2009. ISBN 1586039296.

[76] M. Davis and H. Putnam. A computing procedure for quantification theory. J. ACM, 7(3):201–215, jul 1960. ISSN 0004-5411. doi: 10.1145/321033.321034. URL https://doi.org/10.1145/321033.321034.

[77] M. Awedh and F. Somenzi. Termination criteria for bounded model checking: Extensions and comparison. In BMC@CAV, 2005. URL https://api.semanticscholar.org/CorpusID:7685490.

[78] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design, FMCAD ’00, page 108–125, Berlin, Heidelberg, 2000. Springer-Verlag. ISBN 3540412190.

[79] K.L. McMillan. Interpolation and sat-based model checking. In International Conference on Computer Aided Verification, 2003. URL https://api.semanticscholar.org/CorpusID:11048569.

[80] N. Eén, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In 2011 Formal Methods in Computer-Aided Design (FMCAD), pages 125–134. IEEE, 2011.

[81] A.R. Bradley. Sat-based model checking without unrolling. In Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI'11, page 70–87, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 9783642182747.

[82] X. Zhang, S. Xiao, Y. Xia, J. Li, M. Chen, and G. Pu. Accelerate safety model checking based on complementary approximate reachability. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 42(9): 3105–3117, 2023. doi: 10.1109/TCAD.2023.3236272.

[83] D. Gao and T. Melham. End-to-End Formal Verification of a RISC-V Processor Extended with Capability Pointers. In 2021 Formal Methods in Computer Aided Design (FMCAD), pages 24–33, 2021.

[84] R. Sharafinejad, B. Alizadeh, and T. Nikoubin. Formal Verification of Non-Functional Strategies of System-Level Power Management Architecture in Modern Processors. In 2020 IEEE 14th Dallas Circuits and Systems Conference (DCAS), pages 1–6, 2020. doi: 10.1109/DCAS51144.2020.9330633.

[85] C. Rojas, H. Morales, and E. Roa. A Low-Cost Bug Hunting Verification Methodology for RISC-V-Based Processors. In 2021 IEEE International Symposium on Circuits and Systems (ISCAS), pages 1–5, 2021.

[86] D. Selvakumar, J. Mervin, S. Pattanshetty, and D. Vivian. Formal Verification and Analysis of a Pseudo Random Number Generator. In 2021 25th International Symposium on VLSI Design and Test (VDAT), pages 1–6, 2021. doi: 10.1109/VDAT53777.2021.9601109.

[87] M. Jakobs, F. Pauck, M. Platzner, H. Wehrheim, and T. Wiersema. Software/Hardware Co-Verification for Custom Instruction Set Processors. IEEE Access, pages 1–1, 2021.

[88] L. Duan, Y. Hu, H. Liu, W. Feng, and J. Gan. An Efficient Formal Verification Method in I/O Multiplexing Module Based on VC Formal CC. In 2020 IEEE 3rd International Conference on Electronics and Communication Engineering (ICECE), pages 112–116, 2020. doi: 10.1109/ICECE51594.2020.9353040.

[89] V. Liew, P. Beame, J. Devriendt, J. Elffers, and J. Nordström. Verifying Properties of Bit-vector Multiplication Using Cutting Planes Reasoning. In 2020 Formal Methods in Computer Aided Design (FMCAD), volume 1, pages 194–204. TU Wien Academic Press, 2020.

[90] Y. Xing, H. Lu, A. Gupta, and S. Malik. Leveraging Processor Modeling and Verification for General Hardware Modules, pages 1130–1135. 2021. doi: 10.23919/DATE51398.2021.9474194.

[91] C. Duran, H. Morales, C. Rojas, A. Ruospo, E. Sanchez, and E. Roa. Simulation and Formal: The Best of Both Domains for Instruction Set Verification of RISC-V Based pages 1–4. 2020. doi: 10.1109/ISCAS45731.2020.9180589.

[92] M.R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz. Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking. In 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pages 994–999, 2019. doi: 10.23919/DATE.2019.8715004.

[93] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan. A Formal Approach to Secure Speculation. In 2019 IEEE 32nd Computer Security Foundations Symposium (CSF), pages 288–28815, 2019. doi: 10.1109/CSF.2019.00027.

[94] H. Busch. An automated formal verification flow for safety registers. 2015. URL https://api.semanticscholar.org/CorpusID:20406506.

[95] F.A.D. Silva, A.C. Bagbaba, S. Hamdioui, and C. Sauer. Use of formal methods for verification and optimization of fault lists in the scope of iso26262. 2018. URL https://api.semanticscholar.org/CorpusID:214721167.

[96] F.A. Silva, A.C. Bagbaba, S. Hamdioui, and C. Sauer. Efficient methodology for iso26262 functional safety verification. In 2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS), pages 255–256, 2019. doi: 10.1109/IOLTS.2019.8854449.

[97] F.A. Silva, A.C. Bagbaba, S. Hamdioui, and C. Sauer. Combining fault analysis technologies for iso26262 functional safety verification. In 2019 IEEE 28th Asian Test Symposium (ATS), pages 129–1295, 2019. doi: 10.1109/ATS47505.2019.00024.

[98] F.A. da Silva, A.C. Bagbaba, S. Sartoni, R. Cantoro, M.S. Reorda, S. Hamdioui, and C. Sauer. Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs. In 2020 IEEE European Test Symposium (ETS), pages 1–6, 2020.

[99] F.A. da Silva, A. Cagri Bagbaba, S. Hamdioui, and C. Sauer. An automated formal-based approach for reducing undetected faults in ISO 26262 hardware compliant designs. In 2021 IEEE International Test Conference (ITC), pages 329–333, 2021. doi: 10.1109/ITC50571.2021.00047.

[100] S. Huhn, S. Frehse, R.t Wille, and R. Drechsler. Determining application-specific knowledge for improving robustness of sequential circuits. IEEE Transactions

on Very Large Scale Integration (VLSI) Systems, 27(4):875–887, 2019. doi: 10.1109/TVLSI.2018.2890601.

[101] S. Huhn and R. Drechsler. Next Generation Design For Testability, Debug and Reliability Using Formal Techniques. In 2022 IEEE International Test Conference (ITC), pages 609–618, 2022. doi: 10.1109/ITC50671.2022.00086.

[102] A.C. Bagbaba, M. Jenihhin, R. Ubar, and C. Sauer. Representing gate-level set faults by multiple seu faults at rtl. In 2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS), pages 1–6, 2020. doi: 10.1109/IOLTS50870.2020.9159715.

[103] A. Biere and K. Heljanko. Hardware model checking competition (hwmcc) 2011 benchmarks, 2011.

[104] A. Traskov, T. Ehrenberg, S. Loitz, A. Ayari, A. Efody, J. Hupcey, and Mentor Graphics. Fault proof: Using formal techniques for safety verification and fault analysis. 2016. URL `https://api.semanticscholar.org/CorpusID:250664392`.

[105] A.L.D. Antón, J. Müller, M.R. Fadiheh, D. Stoffel, and W. Kunz. Fault attacks on access control in processors: Threat, formal analysis and microarchitectural mitigation. IEEE Access, 11:52695–52711, 2023. doi: 10.1109/ACCESS.2023.3280804.

[106] .D. Berg and K.A. Label. Verification of triple modular redundancy (tmr) insertion for reliable and trusted systems. 2016. URL `https://api.semanticscholar.org/CorpusID:61468919`.

[107] G. Beltrame. Triple modular redundancy verification via heuristic netlist analysis. PeerJ Comput. Sci., 1:e21, 2015. URL `https://api.semanticscholar.org/CorpusID:16454229`.

[108] L.A.Benites and F.. Kastensmidt. Automated design flow for applying triple modular redundancy (tmr) in complex digital circuits. 2018 IEEE 19th Latin-American Test Symposium (LATS), pages 1–4, 2018. URL `https://api.semanticscholar.org/CorpusID:13850498`.

[109] L. Entrena, A.J. Sánchez-Clemente, L.A. García-Astudillo, M.a Portela-García, M. García-Valderas, A. Lindoso, and R. Sarmiento. Formal verification of fault-tolerant hardware designs. IEEE Access, 11:116127–116140, 2023. URL `https://api.semanticscholar.org/CorpusID:264338901`.

[110] Muth. A nine-valued circuit model for test generation. IEEE Transactions on Computers, C-25(6):630–636, 1976. doi: 10.1109/TC.1976.1674663.

[111] U. Krautz, M. Pflanz, C. Jacobi, H.W. Tast, K. Weber, and H.T. Vierhaus. Evaluating coverage of error detection logic for soft errors using formal methods. In Proceedings of the Design Automation Test in Europe Conference, volume 1, pages 1–6, 2006. doi: 10.1109/DATE.2006.244062.

[112] V. Paruthi, C. Jacobi, and K. Weber. Efficient symbolic simulation via dynamic scheduling, don't caring, and case splitting. In Conference on Correct Hardware Design and Verification Methods, 2005. URL `https://api.semanticscholar.org/CorpusID:16009754`.

[113] J. Schreiner, R. Findenigy, and W. Ecker. Design centric modeling of digital hardware. In 2016 IEEE International High Level Design Validation and Test Workshop (HLDVT), pages 46–52, 2016. doi: 10.1109/HLDVT.2016.7748254.

[114] K. Devarajegowda and W. Ecker. On generation of properties from specification. In 2017 IEEE International High Level Design Validation and Test Workshop (HLDVT), pages 95–98, 2017. doi: 10.1109/HLDVT.2017.8167470.

[115] M.R. Fadiheh, J. Urdahl, S.S. Nuthakki, S. Mitra, C. Barrett, D. Stoffel, and W. Kunz. Symbolic quick error detection using symbolic initial state for pre-silicon verification. In 2018 Design, Automation Test in Europe Conference Exhibition (DATE), pages 55–60, 2018. doi: 10.23919/DATE.2018.8341979.

[116] K. Devarajegowda, M.R. Fadiheh, E. Singh, C. Barrett, S. Mitra, W. Ecker, D. Stoffel, and W. Kunz. Gap-free processor verification by s2qed and property generation. In 2020 Design, Automation Test in Europe Conference Exhibition (DATE), pages 526–531, 2020. doi: 10.23919/DATE48585.2020.9116515.

[117] H. Liu, J. Yin, C. Huang, H. Lan, Z. Jin, Z. Zheng, and X. Zhang. A fault injection and formal verification framework based on uml sequence diagrams. In 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), pages 45–50, 2023. doi: 10.1109/ISSREW60843.2023.00045.

[118] C.R. Lincoln. Specifying systems: The tla+ language and tools for hardware and software engineers. Software Quality Professional, 5(4):43–43, 2003.

[119] W.E. Vesely, F.F. Goldberg, N. Roberts, and D.F. Haasl. Fault Tree Handbook. NRC, 1987. URL `https://api.semanticscholar.org/CorpusID:60773312`.

[120] M. Ammar, G.B. Hamad, O.A. Mohamed, and Y. Savaria. Towards an accurate probabilistic modeling and statistical analysis of temporal faults via temporal dynamic fault-trees (tdfts). IEEE Access, 7:29264–29276, 2019. doi: 10.1109/ACCESS.2019.2902796.

[121] A. Samadi, M. Ammar, and O.A. Mohamed. Dynamic Fault Tree Analysis and Risk Mitigation Strategies of Data Communication System via Statistical Model Checking. In 2021 19th IEEE International New Circuits and Systems Conference (NEWCAS), pages 1–4, 2021. doi: 10.1109/NEWCAS50681.2021.9462743.

[122] K.D. Rao, V. Gopika, V.S. Rao, H.S. Kushwaha, A.K. Verma, and A. Srividya. Dynamic fault tree analysis using monte carlo simulation in probabilistic safety assessment. Reliability Engineering & System Safety, 94(4):872–883, 2009.

[123] Q. Shao, S. Yang, and X. Gou. Formal Analysis of Multiple-Cell Upset Failure Based on Common Cause Failure Theory. IEEE Transactions on Reliability, 70 (4):1495–1509, 2021. doi: 10.1109/TR.2020.3010937.

[124] S. Frehse, G. Fey, A. Suflow, and R. Drechsler. Robustness Check for Multiple Faults Using Formal Techniques. In 2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, pages 85–90, 2009. doi: 10.1109/DSD.2009.218.

[125] R. Leveugle. A new approach for early dependability evaluation based on formal property checking and controlled mutations. In 11th IEEE International On-Line Testing Symposium, pages 260–265, 2005. doi: 10.1109/IOLTS.2005.8.

[126] I. Buzhinsky and A. Pakonen. Model-Checking Detailed Fault-Tolerant Nuclear Power Plant Safety Functions. IEEE Access, 7:162139–162156, 2019. doi: 10.1109/ACCESS.2019.2951938.

[127] A. Pakonen and I. Buzhinsky. Verification of fault tolerant safety IC systems using model checking. In 2019 IEEE International Conference on Industrial Technology (ICIT), pages 969–974, 2019. doi: 10.1109/ICIT.2019.8755014.

[128] E. Segev, S. Goldshlager, H. Miller, O. Shua, O. Sher, and S. Greenberg. Evaluating and comparing simulation verification vs. formal verification approach on block level design. In Proceedings of the 2004 11th IEEE International Conference on Electronics, Circuits and Systems, 2004. ICECS 2004., pages 515–518, 2004. doi: 10.1109/ICECS.2004.1399731.

[129] S. Verma, P. Lee, and I.G. Harris. Error Detection Using Model Checking vs. Simulation. In 2006 IEEE International High Level Design Validation and Test Workshop, pages 55–58, 2006. doi: 10.1109/HLDVT.2006.319964.

[130] A.J. Hu. Simulation vs. Formal: Absorb What Is Useful; Reject What Is Useless. In Karen Yorav, editor, Hardware and Software: Verification and Testing, pages 1–7, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-77966-7.

[131] W.K. Lam.
Hardware Design Verification: Simulation and Formal Method-Based Approaches. Prentice Hall PTR, USA, 1st edition, 2008. ISBN 0137010923.

[132] T. Mancini, F. Mari, A. Massini, I. Melatti, F. Merli, and E. Tronci. System Level Formal Verification via Model Checking Driven Simulation. In Natasha Sharygina and Helmut Veith, editors, Computer Aided Verification, pages 296–312, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39799-8.

[133] R. Lipka, M. Paška, and T. Potužák. Simulation Testing and Model Checking: A Case Study Comparing these Approaches. In István Majzik and Marco Vieira, editors, Software Engineering for Resilient Systems, pages 116–130, Cham, 2014. Springer International Publishing. ISBN 978-3-319-12241-0.

[134] L. Lavagno, I. Markov, G. Martin, and L. Scheffer. Electronic Design Automation for IC System Design, Verification, and Testing, 2016.

[135] M. Girish, G. Gopakumar, and D.S. Divya. Formal and Simulation Verification: Comparing and Contrasting the two Verification Approaches. In 2021 2nd International Conference on Advances in Computing, Communication, Embedded and Secure Systems (ACCESS), pages 41–44, 2021. doi: 10.1109/ACCESS51619.2021.9563305.

[136] IEEE Standard for Verilog Hardware Description Language. IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), pages 1–590, 2006. doi: 10.1109/IEEESTD.2006.99495.

[137] RISC-V Instruction Set Manual, Volume II: Privileged Architecture, .

[138] R. Barbosa, A. Fonseca, and F. Araujo. Reductions and abstractions for formal verification of distributed round-based algorithms. Software Quality Journal, 29 (3):705–731, Sep 2021. ISSN 1573-1367. doi: 10.1007/s11219-020-09539-6. URL `https://doi.org/10.1007/s11219-020-09539-6`.

[139] H. Witharana, Y. Lyu, S. Charles, and P. Mishra. A Survey on Assertion-Based Hardware Verification. ACM Comput. Surv., 54(11s), sep 2022. ISSN 0360-0300. doi: 10.1145/3510578. URL `https://doi.org/10.1145/3510578`.

[140] T. Yamaguchi, B. Hoxha, D. Prokhorov, and J.V. Deshmukh. Specification-guided Software Fault Localization for Autonomous Mobile Systems. In 2020 18th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE), pages 1–12, 2020. doi: 10.1109/MEMOCODE51338.2020.9315067.

[141] P.A. Patil and C. Kulkarni. A Survey on Multiply Accumulate Unit. In 2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA), pages 1–5, 2018. doi: 10.1109/ICCUBEA.2018.8697705.