Verification Framework for a RISC-V Processor based on Coverage-Guided Fuzzing

Masterarbeit
zur Erlangung des akademischen Grades
Master of Science

vorgelegt von
Ashvin Vaidyanathan

Institute for Embedded Systems
Technische Universität Hamburg

Jan 2025

1st Examiner: Prof. Dr. Görschwin Fey
2nd Examiner: Prof. Dr. Ulf Kulau

Contents

Statutory Declaration

I, Ashvin Vaidyanathan, herewith formally declare that I have authored the submitted master thesis titled ***Verification Framework for a RISC-V Processor based on Coverage-Guided Fuzzing*** independently.

I did not use any outside support except the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and sources which I employed when producing this academic work, either literally or in content. The work has not been submitted in the same or similar form to any other examination office. I agree to the dissemination of my work for scientific purposes.

I am aware that the violation of this regulation will lead to failure of the thesis.

Signature, June 25, 2025

Acknowledgment

I am deeply grateful to Prof. Dr. -Ing. Görschwin Fey for his guidance and supervision throughout this thesis. His expertise in Electronic Design Automation and embedded systems, along with his deep understanding of design and automation in Cyber-physical systems, provided a solid foundation for my research on the Verification Framework for a RISC-V Processor based on Coverage-Guided Fuzzing. His thoughtful feedback and consistent support played a crucial role in shaping this work, and I truly appreciate his patience, encouragement, and dedication to fostering an environment of learning and growth. It has been an honor to work under his supervision.

I would like to express my gratitude to Prof. Dr. Ulf Kulau, Smart Sensors Group, Technische Universität Hamburg who kindly agreed to serve as the second supervisor for this thesis. His extensive knowledge and prior work in integration and evaluation of RISC-V software core have significantly influenced the direction and scope of this research. I deeply appreciate his support in overseeing this work.

Lastly, I would like to thank my family and friends for their constant care and support throughout the thesis work.

Abstract

Increasing complexity of modern processors, has increased the need for improved verification techniques now more than ever. In this thesis, a verification framework is proposed for a RISC-V processor using the novel coverage-guided fuzzing methodology, that is slowly gaining popularity in the hardware community after seeing its benefits in the software side.

The framework combines a test generation module, a simulation model that handles both the Device-under-Test(DUT) and the Golden Reference Model(GRM), a dynamic coverage feedback module and most importantly, a fuzz engine that allows the test cases to be target rare instructions, and complex branches. The framework also integrates a detailed logging mechanism that allows for both ral-time coverage updates as well as help identify and categorize any potential bugs.

The framework is also built using only open-source software, keeping with the spirit of RISC-V's open source beginnings. The thesis aims to highlight the importance of coverage-guided fuzzing in the field of hardware verification while building a verification framework that is modular, efficient, and effective.

List of Abbreviations

AFL	American Fuzzing Loop.
ASICs	Application-Specific Integrated Circuits.
CDG	Coverage Directed Test Generation.
CR	Constrained Random.
DUT	Device Under Test.
FPGAs	Field-Programmable Gate Arrays.
FSM	Finite State Machine.
GRM	Golden Reference Model.
HDL	Hardware Description Language.
HSB	Hardware Simulation Binary.
HW	Hardware.
IPs	Intellectual Property.
ISA	Instruction Set Architecture.
ISS	Instruction Set Simulator.
NOPs	No Operations.
OVL	Open Verification Library.
PC	Program Counter.
PIC	Position Independent Code.
PK	Proxy Kernel.
PSL	Property Specification Language.
PUT	Program Under Test.
RTL	Register Transfer Level.
SIPs	Soft IPs.
SW	Software.
UVM	Universal Verification Methodology.
VCD	Value Change Dump.

1 Introduction

RISC-V is an Instruction Set Architecture (ISA) conceptualized and designed in the year 2010 at the University of California, Berkeley[1]. The RISC-V ISA supports computer architecture design and education but has over time become a standard free and open architecture for industry implementations as well. This kind of an open-source ISA has many advantages over the commercial ISAs:

1. Commercial ISAs are proprietary, which makes it very hard for groups willing to share actual Register Transfer Level (RTL) implementation, be it at an academic level or at a business level.

2. Popular ISAs are very complex, and much of the complexity is a result of an outdated design decisions and legacy code that can no longer be modified, rather than features that actually increase efficiency.

3. Commercial ISAs come and go and many ISAs that were quite popular years ago are no longer even in production, which leaves third parties unable to continue supporting the ISA due to lingering intellectual property issues. While an open-source ISA may lose popularity, any interested party can continue using and developing the ecosystem.

4. Commercial ISAs were not designed for extensibility, and as a consequence have added considerable instruction encoding complexity as their instruction sets have grown(as can be seen with ISAs like ARM).

RISC-V has received significant support from the open-source community and has seen a huge rise in adoption by the industry leaders. In its list of members are big companies like Google, NVIDIA, Seagate, Qualcomm, Intel and AMD[2]. As of 2024, NVIDIA alone is shipping 1 Billion RISC-V cores worldwide as part of their chips[3].

Open-source cores, however, come with significant verification challenges. Verification is already a bottleneck for the design cycles, with some surveys stating that validation, verification, and testing take up 60% of the total production cost[4]. Companies approaching open-source IPs usually verify their functionality internally and can provide reports or fixes to the Intellectual Property (IPs) designers to improve the quality of free hardware. However, a completely open-source verification framework for open-hardware would allow free access to the verification efforts and would increase the reliability of free IPs[4]. With the rising popularity of RISC-V, the complexity of the ISA keeps increasing, and while the addition of features is a very well-monitored affair, the time and resources required for comprehensive verification.

Clifford Wolf from SymbioticEDA developed an open-source end-to-end formal verification framework for RISC-V processors[5]. However, it requires changes in the Device Under Test (DUT) interfaces to be integrated into the framework. While there have been efforts from companies like Imperas to define a standardized verification interface[6] which will allow all RISC-V based DUTs to have the same interface thereby allowing the re-use of testbenches across different teams and even different companies, as of the time this thesis was written not many processors outside of the Imperas and OpenHW group have adopted this interface.

Despite these efforts, the task of achieving a comprehensive verification for RISC-V processors is still a complex challenge due to the modular and customizable nature of the ISA. While this provides a level of flexibility on the developer's side that is not available in other commercial ISAs to create custom instructions and optimizations, it also makes the verification process complicated. Different implementations of the ISA that are designed with different extensions introduce a lot of edge cases and sources of errors that cannot always be addressed by traditional dynamic verification methodologies. This has led to an increasing demand for open-source verification frameworks that are adaptable and scalable while being able to handle any custom features native to the design.

Also, the lack of a universally adopted verification standard or interface makes it difficult for design teams to share verification assets like testbenches or coverage models. Each implementation needs to be tested in isolation with a large part of the verification assets being built from scratch, taking up a lot of time and resources. A robust and flexible open-source framework, capable of leveraging state-of-the-art verification techniques like mutation and fuzz testing is essential to keep the advancement of RISC-V steady.

1.1 Motivation

Functional verification is not a new topic and has a solid standard that has been set up even for the relatively new RISC-V ISA. Most available formal verification and testing frameworks are built using Universal Verification Methodology (UVM), which assumes access to commercial simulation tools and personnel is not a limitation. Small design teams, individuals, or students, however, usually do not have the budget or tools to tackle the verification problem using this approach. The alternative here is to approach this problem using open-source verification tools where possible and optimize the hardware and software resources[7]. Even verification solutions that are described as open-source usually involve the use of some commercial tool like Cadence or Synopsys for simulation[7],[8] which may not be accessible to everyone.

While formal verification shows promising results for testing high-quality designs[9], dynamic verification is still widely used due to its accessibility and ability to trigger more corner-cases that would otherwise not be accessed. These corner cases are usually defined as part of the coverage metrics used for the verification process. Once the coverage metrics are defined, they are used to measure the quality of the test inputs used for dynamic verification. The natural next step in this process is to automate the stimuli generation while maximizing coverage. If we use the feedback of the coverage criteria to generate further inputs, it is called Coverage Directed Test Generation (CDG). Coverage-guided mutational fuzz testing has become a popular testing technique for finding vulnerabilities in software systems[9].

In the hardware domain, CDG techniques are deployed using coverage metrics like HDL line, Finite State Machine (FSM) coverage, functional coverage etc in a feedback loop to generate tests that increase state space exploration[10]. Compared to formal techniques such as symbolic execution, fuzz testing has been able to scale up to much bigger real-world programs with smaller setup and engineering efforts. Although this method seems very promising, there has not been widespread adoption in hardware-based dynamic verification, due to some key challenges resulting from differences in hardware and software execution models[9].

Hardware designs are usually written in RTL code which is not inherently executable like software code. These RTL codes need to be simulated after being translated to a software model and combined with a design-specific testbench and simulation engine, all of which together form a Hardware Simulation Binary (HSB). This introduces a lot of additional complexity as well as

computational effort.

Secondly, unlike most software, hardware testing requires sequences of structured inputs (which are tailored to the DUT) to drive meaningful state transitions. Finally, integrating fuzzing tools into such a setup often requires extensive modification to the DUT interfaces to handle both fuzzing and simulation environments cohesively. This again means a lot of additional time and effort.

1.2 Tools used

As mentioned in 1.1 and 1.3, a key aspect of the thesis is to build the verification framework using only open-source tools. This meant that instead of the more popular commercial simulators offered by vendors like Synopsys, Cadence or Siemens, in this thesis open-source simulators like Verilator and Icarus Verilog for simulation and debugging purposes. A detailed analysis of the tools used, why they were selectred and their advantages and disadvantages are elaborated upon in section 3.6.

1.3 Objectives

- The primary objective of this thesis is to set up a testing framework for a RISC-V processor core based on coverage-guided fuzz testing. The aim of the framework is to generate test cases based on coverage feedback to explore previously untested areas of the processor design.
- The coverage criteria for the tests will be defined based on structural and functional coverage. Structural coverage will target the key elements such as statement, branch and toggle coverage. Functional coverage will focus on features like instruction set, register and immediate value coverage. This two-pronged approach seeks to provide a comprehensive assessment of the selected processor's reliability.
- One of the main goals is to automate the test generation-coverage analysis-fuzzing feedback loop in order to reduce the manual effort needed to test the DUT and require minimal intervention.
- The entire framework is built ground up while trying to keep it as design-agnostic as possible to the DUT. This means the instruction generator, the coverage calculator and the fuzz engine are written so that they can be used for any other DUT with minimal to no changes.
- Finally, the entire framework will be set-up using only completely open-source software including the simulator to respect the founding spirit of the RISC-V ISA.

1.4 Outline

The thesis is divided into six broad chapters.

The first chapter (this one), goes into the motivation behind the thesis and what objectives it aims to complete.

The second chapter provides all the necessary background about this topic that will allow readers who are not fully in touch with this subject come up to speed. This chapter also covers

the very important related work segment in which the work of all the others' who have made progress in this field is presented.

The third chapter is for the methodology of the thesis. It clarifies the design philosophies and provides reasoning for why certain decisions were taken . It also provides a conceptual look into the structure of the thesis without getting bogged down by the minutiae of implementation.

The fourth chapter is implementation, where every aspect of the thesis' implementation is given in detail. From setting up for the first simulation to logging and analysing the last detail is explained here.

In the fifth chapter, the results are discussed, and apart from the normal outputs, particularly interesting results that showed up during the thesis are shown here and have a brief discussion from the author.

Finally, in the sisth chapter the author concludes on their work and presents what future work they have envisioned for their thesis.

The source code for this thesis can be found at: https://github.com/AshvinVaidyanathan/masterthesis

2 Background and Related Work

2.1 The RISC-V ISA Overview

The RISC-V ISA was started in May 2010 by Prof. Krste Asanovic and his two graduate students Andrew Waterman and Yunsup Lee as a part of the Parallel Computing Laboratory at UC Berkeley, of which Prof. David Patterson was Director[11]. The goal for RISC-V is to become a universal ISA[12]:

- It should suit all sizes of processors, i.e., from the tiniest embedded controller to the fastest high-performance computer.
- It should work well with a wide variety of popular software stacks and programming languages.
- It should accommodate all popular implementation technologies:Field-Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), full-custom chips and even any future device technologies.
- It should be efficient for all micro-architecture styles: microcoded or hardwired control; in-order, decoupled, or out-of-order pipelines; single or superscalar instruction issue; and so on.
- It should support extensive specialization to act as a base for customized accelerators, which rise in importance as Moore's Law fades.
- It should be stable, in that the base ISA should not change. More importantly the ISA cannot be discontinued as happened with proprietary ISAs like the AMD Am29000, Intel i860, Intel i960, Motorola 88000 etc.

RISC-V is unusual; not only because it was designed recently - whereas most of it's competitors date from the 1960s - but also because it is an open ISA. Unlike most proprietary ISAs, its future is not controlled by a single corporation, but instead belongs to a open, non-profit foundation. Founded in 2015, the RISC-V Foundation has more than 325 members, both from academia and corporations. Members of the RISC-V foundation have access to and participate in the development of RISC-V ISA specifications and the related hardware/software (HW/SW) ecosystem[2]. The goal of the RISC-V foundation is to maintain the stability of RISC-V and help it evolve slowly and carefully. Table 1 lists the largest corporate members of the RISC-V Foundation.

2.1.1 A Modular ISA

Conventionally, ISAs are designed in an incremental fashion. This means that anytime a new processor is designed, it must be able to run not only the new ISA extensions but also all the ISA extensions from the past. This ensures backwards binary-compatibility, which means that

>$50B	5B−50B	0.5B−5B
Google Huawei IBM Microsoft Samsung	BAE Systems MediaTek Micron Tech. NVIDIA NXP Semi. Qualcomm Western Digital	AMD Andes Technology C-SKY Microsystems Integrated Device Tech. Mellanox Technology Microsemi Corp.

Table 2.1: The corporate members of the RISC-V Foundation and their annual sales as of the 6th RISC-V Workshop in May 2017. The left column companies all exceed $US 50B in annual sales, the middle column companies sell less than $US 50B but more than $US 5B and the companies in the right column sell less than $US 5B but more than $US 0.5B. The Foundation also includes many smaller companies, start-up companies, non-profit organizations and universities[12].

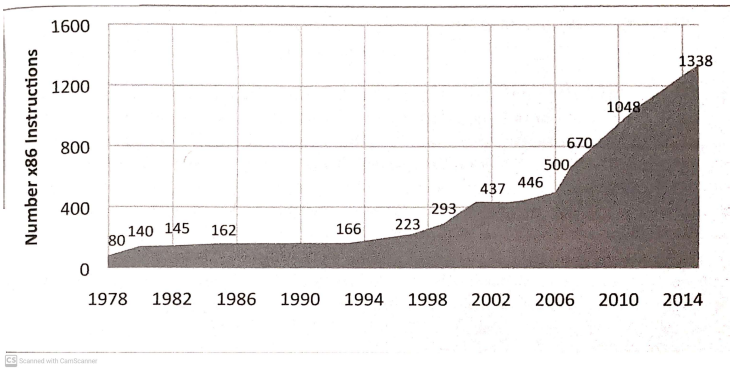


Figure 2.1: Growth of x86 ISA over its lifetime. This graph is a conservative estimate. An Intel blog[13] puts the estimate at 3600 instructions at 2015[12].

even the newest processors can run binary code from decades old extensions of the ISA. While this is important for certain instructions, it is not needed for many of the instructions as the newer extensions were built to be more efficient and effective. This leads to the ISA becoming very large and bulky.

A good example of this is the 80x86 ISA as seen in Fig. 2.1. It was initially designed to be a stopgap implementation to counter Zilog. Over time, however, it ended up being the primary processor from Intel and the x86 remained the underlying ISA for many of Intel's popular products. The x86 dates back to 1978 and since then has added about *three instructions per month* over its lifetime.

The RISC-V ISA on the other hand is *modular*. It has a core *RV32I* base ISA that is frozen and will never change. This gives compiler writers, assembly language programmers, and operating system developers a stable target. On top of this base ISA, optional standard extensions can be included for the hardware implementation. This modularity enables very small and low-energy applications of RISC-V which is critical for embedded applications. By telling the compiler which extensions are included, it can create targeted code which is best suited for the hardware implementation. For e.g., RV32IMFD adds:

- RV32M - Multiply
- RV32F - Floating-point

Name	Description	Status	Instruction Count
RV32I	Base integer Instruction Set - 32bit	Frozen	49
RV32E	Base integer Instruction Set - 32-bit	Open	49
RV64I	Base integer Instruction Set - 64-bit	Frozen	14
RV128I	Base integer Instruction Set - 128-bit	Open	14
Extension			
M	Multiplication and Division	Frozen	8
A	Atomic Instructions	Frozen	11
F	Single-Precision Floating-Point	Frozen	25
D	Double-Precision Floating-point	Frozen	25
G	Shorthand for the base and above extensions	n/a	n/a
Q	Quad-Precision Floating-Point	Frozen	27
L	Decimal Floating-Point	Open	Undefined Yet
C	Compressed Instructions	Frozen	36
B	Bit Manipulation	Open	42
J	Dynamically Translated Languages	Open	Undefined Yet
T	Transactional Memory	Open	Undefined Yet
P	Packed-SIMD Instructions	Open	Undefined Yet
V	Vector operations	Frozen	186
N	User-Level Interrupts	Open	3
H	Hypervisor	Frozen	2
S	Supervisor-Level Instructions	Open	7

Table 2.2: Standard extensions for the RISC-V ISA that enable additional functionality and can be included in a modular fashion based on the hardware and the code[14].

- RV32D - Double-precision floating point

The table 2.2 shows the standard extensions beyond the base instructions which can be implemented based on design goals.

RISC-V also defines an exact order that must be used to define the RISC-V subset:

```
RV [32, 64, 128] I, M, A, F, D, G, Q, L, C, B, J, T, P, V, N
```

For example, RV32IMAFDQC is legal, whereas RV32IMFDACQ is not.

2.1.2 Design Philosophy

The RISC-V ISA was designed keeping the following seven principles in mind. To illustrate the thought process, some comparisons with older, more popular ISAs are shown by[12]:

1. **Cost.** Processors are implemented as ICs and the cost of creating a processor is very sensitive to the area of the die, i.e., $cost \approx f(diearea^2)$. An architect, therefore, wants to keep the ISA simple to shrink the size of the processor. The RISC-V ISA is much simpler than the ARM-32 ISA. As a concrete example of simplicity, on comparing a RISC-V Rocket processor to an ARM-32 Cortex-A5 processor in the same technology (TSMC40GPLUS) using the same sized caches (16 KiB), the RISC-V die is 0.27mm^2 versus 0.53mm^2 for ARM-32.

2. **Simplicity.** As seen before, ISA simplicity reduces cost of fabrication in lieu of reducing die area. ISA simplicity also reduces the cost of design and verification time. As an example of ISA complexity this instruction from ARM-32:

```
ldmiaeq SP!, {R4-R7, PC}
```

   This instruction stands for LoaD Multiple Increment-Address on eQual. It performs 5 data loads and writes to 6 registers but only executes is the EQ condition is set. Moreover, it writes the result to PC, so it is also performing a conditional branch. However, compilers usually prefer simple instructions to complex ones. So if a complex instruction can be replaced by more than one simple instructions, that is most likely to be executed.

3. **Performance.** Other than embedded microprocessors, architects are also concerned about performance as well as cost. Performance is usually given by

$$\frac{instructions}{program} \times \frac{avg.clockcycles}{instructions} \times \frac{time}{clockcycle} = \frac{time}{program} \tag{2.1}$$

   Even if a simple ISA executes more instructions per program, it can make up for it by having faster clock cycles or fewer clock cycles per instructions. For e.g., the CoreMark benchmark [15, 12], the performance on the ARM-32 Cortex-A9 is:

$$\frac{32.27Binsn}{program} \times \frac{0.79clockcycles}{instructions} \times \frac{0.71ns}{clockcycle} = \frac{18.15s}{program} \tag{2.2}$$

   For the BOOM implementation of RISC-V it is:

$$\frac{29.51Binsn}{program} \times \frac{0.72clockcycles}{instructions} \times \frac{0.67ns}{clockcycle} = \frac{14.26s}{program} \tag{2.3}$$

4. **Isolation of the Architecture from Implementation.** Architecture is what a machine language programmer needs to know to write a correct program, but not the performance of that program. There is a temptation for an architect to include instructions in the ISA that help a particular implementation at a particular time but can hinder different or future implementations. For e.g., ARM-32 and some other ISAs have a Load Multiple instruction which can improve performance of a single-instruction issue pipelined designs, but can hurt multiple instruction-issue pipelines.

5. **Room for Growth.** With the ending of Moore's Law in sight, the path forward for major improvements is using custom instructions for specific domains. This means it is important to reserve opcode space for future enhancements. In the 70s and 80s, when Moore's Law was in full force, there was little thought of saving opcode space for future accelerators. Architects instead valued larger address and immediate fields to reduce instructions per program. An example, was when the architects of the ARM-32 later tried to reduce code size by adding 16-bit length instructions to the formerly uniform 32-bit length ISA, but there was no room left. The only solution was to create a new ISA (Thumb and Thumb-2) and introduce a mode to switch between ISAs.

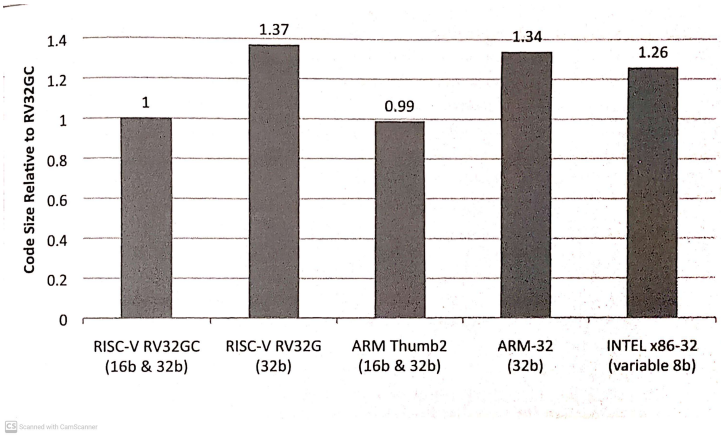


Figure 2.2: Relative program sizes for RV32G, ARM-32, x86-32, RV32C, and Thumb-2. RV32G indicates a popular combination of RISC-V extensions, properly called RV32IMAFD[12].

6. **Program Size.** Smaller the program, smaller the area on the chip needed for program memory. Smaller programs also leads to fewer misses in instruction caches, which saves power. Small code size is therefore very important to ISA architects. The x86 ISA has instructions as short as 1 byte and as long as 15 bytes. As shown in Fig 2.2, both ARM-32 and RV32G programs are 6-9% larger than x86 code. However, going against this logic, x86 instructions are also 26% larger than RV32C and Thumb2 instructions which offer 16-bit and 32-bit instructions.

7. **Ease of programming, linking and compiling.** Data in a register is much easier to access than data in memory. This task is easier when there are more registers than not. ARM-32 has 16 registers and x86-32 has 8, while RISC-V has 32 integer registers. Another issue is figuring out the speed of a code sequence. Most RISC-V instructions take one cycle per instructions *barring cache misses), while ARM-32 and x86-32 instructions take multiple clock cycles even without cache misses. This makes performance predictability of the processor difficult for designers. Finally, it is very useful for ISAs to support Position Independent Code (PIC), because it supports dynamic linking. PC-relative branches are a boon to PIC. RISC-V provides PC-relative branches, while x86-32 and MIPS-32 omit this feature.

2.2 Hardware Verification

Verification is the activity that determines the correctness of the design that is being created. It ensures that the design functions according to specification and operates properly. A designer cannot always think of all the different possibilities and scenarios their design will be put through. If the product is sent out without verification, it can cause significant loss of finances and a damage to the company's reputation. Verification is the process by which the design is put through a variety of situations that it may encounter at the hands of the user and ensures that it does not malfunction.

It is not unusual for a chip of reasonable complexity to undergo multiple tape-outs before mass production. While the role of a design engineer ends with having translated the specification

into an implementation that meets the architectural specifications, a verification engineer must verify the design under all cases, multiple time during the design process[16].

During the implementation, a human reads the specification and converts it into a functioning design, usually in the form of some RTL code. that is then transformed into the design. This process is open to the possibility of misinterpretation or omissions due to the involvement of multiple people. Broadly speaking, there are two ways to verify any design: functional verification, in which the process of verification can be primarily accomplished by placing the DUT in a testbench and then applying some test vectors to the design to ensure that it meets specification. There are multiple ways to approach this based on how much of the DUT is involved during the test generation process. The second is formal verification, in which the methodology is to prove that the intent of the design is met using a mathematical approach than the simulation-based approach.

2.2.1 Functional Verification

- **Black Box Approach**. In this method, no information about the DUT's internal designs are taken into consideration. Stimuli are applied to the inputs of the DUT and its outputs are monitored. The pass and fail criteria are determined by looking at the outputs for a certain input and determining its correctness based on the specification. While this approach is suitable for picking out behavioral errors in the earlier stages of design, it soon loses viability as it is not possible to isolate the problem when a test fails. Since, the tests are also not written with the DUT's design in mind, they are highly reusable.
- **White Box Approach.** In this method, there is detailed knowledge about the DUT's internal design. This ensures that verification is possible until the lowest level. The tests can be written to exercise every single block of the design and find bugs faster. The downside is that every testbench and stimuli are almost tailor-made for the DUT and there is little scope of reusability.
- **Grey Box Approach.** This is a middle-ground approach that takes the pros of both Black and White Box approaches as seen in Fig. 2.3. The engineer makes use of their knowledge of the general architecture of the design. While there is minimal access to intermediate points of a block, which handle inter-block communications, the inputs and outputs of the blocks themselves are made available which makes monitoring the blocks easy.

There are many different ways in which simulation-based or functional verification can be approached, depending on how the stimuli are generated, how the input state-space is explored, how the pass/fail criteria are checked etc. Some of the most popular ones according to [16] are:

1. **Instruction Driven Verification** functional verification of devices including a microprocessor have always been very challenging as the ISA compounds the size of the input state space largely. In this approach, the stimuli are usually coded in C or assembly. An automatic test generator is usually built which has knowledge of the processor's ISA and it then outputs a stream of bits or a testcase which is used to validate the DUT. It is also possible to augment these test cases with hand-written tests that the verification engineer wants to make sure are tested.
2. **Random Testing**A typical random generator runs in parallel to the simulator, with various constraints and the device's architectural features as input to generate test cases

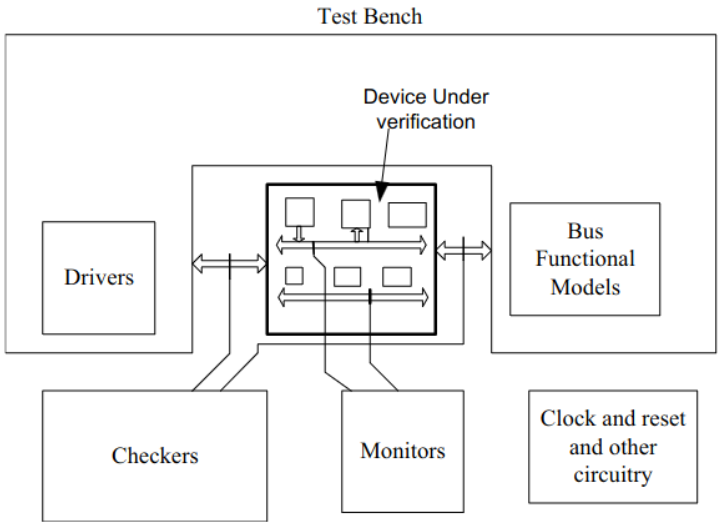


Figure 2.3: White and Grey Box Approaches to Hardware Verification[16]

of interest to the DUT. This helps uncover bugs faster than DUT because it does not depend on the engineer's knowledge of the design.

3. **Coverage Driven Verification** Coverage Driven verification works by collecting coverage data during simulation via assertions or other mechanisms. The new tests that are generated are done so in an attempt to cover areas of the DUT that were not previously covered. This helps the verification reach functional closure faster than directed tests.

4. **Golden Model Approach** In this method a Golden Reference Model (GRM) is typically used to determine the pass or failure of the test. A test sequence is passed through both the RTL and the reference model. The results from both simulations are compared with one another at some pre-determined points. The test case is considered to have passed if the results from golden model and design are in complete agreement. The benefit of a GRM is that it provides an automatic checking mechanism. However, the GRM needs to be developed for the DUT if it does not exist, and also needs to be verified itself.

5. **Pre-Post Processing Approach** This approach is best suited for complex programs where it is difficult to set up a self-checking simulation. In this approach, a script of program generates input data and saves it in a file. this data is then fed into the testbench and from there the DUT. The data collected from the DUT during the simulation is then fed into another script which then determines the pass/fail of tests.

6. **Assertion Based Verification** This is one of the newer forms of verification. In this methodology, the verification engineer embeds *assertions* into various portions of the design to speed up the verification process. An assertion is a concise description of a complex or expected behavior. Assertions are specified using languages like Property Specification Language (PSL) or using Open Verification Library (OVL) modules, SVA or other approaches. Many vendors have their proprietary formats. SystemVerilog/Universal Verification Methodology (SV/UVM) is the most popular in the market[17].

It is important to note that the approaches mentioned here are not exclusive to each other.

Verification teams usually build their own verification methodology on a project-to-project basis depending on the design and the verification flow.

2.2.2 Formal Verification

Formal Verification, in contrast to testing uses rigorous mathematical reasoning to show that a design meets all or parts of its specification. A pre-requisite for the applicability of formal verification is the existence of formal descriptions for both the specifications and the design. Such a description is given in a notation with formal semantics which unambiguously associates a mathematical object with the description, permitting these objects to be reasoned about in a mathematical framework. The time required for formal verification needs to be considered when applying these techniques to a real-life project. Applying formal verification frameworks to large state-of-the-art designs requires large amounts of time of highly skilled experts. Although formal verification methods have been used in the verification of many state-of-the-art microprocessors and other complex chips, no top-to-bottom formal verification has been implemented for designs of such complexity. The cost of such a process is still prohibitive[18].

2.3 Fuzz Testing

A Fuzz tester or a *fuzzer* is a tool that iteratively and randomly generates inputs with which it tests a target program. Despite appearing "naive" when compared to more sophisticated tools involving SMT solvers, symbolic execution, and static analysis, fuzzers are surprisingly effective[19]. The first fuzzers appeared in the early 1990s, primarily relying on some forms of blackbox testing. In this case, providing the Program Under Test (PUT) with randomly generated inputs, with crashes and errors as the only guidelines for the fuzzing campaigns. The popularity of fuzzing in real-world software systems has made it also interesting in the field of hardware verification, especially when used in frameworks that work with CDG[9].

Most modern fuzzers follow the procedure outlined in Fig.2.4. The process begins by choosing a corpus of "seed" inputs with which to test the target program. The fuzzer then repeatedly mutates these inputs and evaluates the program under test. If the result produces "interesting" behavior, the fuzzer keeps the mutated input for future use and records what was observed. Eventually the fuzzer stops, either due to reaching a particular goal (e.g., finding a certain sort of bug) or reaching a timeout.

Over the past 5 years, both industrial and academic research on fuzz testing has reached a consensus on a *de facto* standard for fuzzing - the American Fuzzing Loop (AFL) released in 2013 by Michael Zalewski[20].

2.3.1 Coverage-Guided Fuzz Testing

In electronic design, a semiconductor intellectual property (IP) core is a reusable unit of logic, cell or IC layout design. Soft IPs (SIPs) are IP cores generally offered as synthesizable RTL modules developed in an HDL like SystemVerilog. A SIP implements a specific logic that is concise enough to be tested using fuzzing techniques while not running into scalability issues[21]. Fuzzing stimuli to SIP Verilog modules in a simulation environment is challenging largely because the simulation is slow. Even if test cases are generated in parallel, simulation needs to happen in parallel, along with enabling coherency between different stimuli generators to avoid repetition of testcases.

Core fuzzing algorithm:

```
corpus ← initSeedCorpus()
queue ← ∅
observations ← ∅
while ¬isDone(observations,queue) do
    candidate ← choose(queue, observations)
    mutated ← mutate(candidate,observations)
    observation ← eval(mutated)
    if isInteresting(observation,observations) then
        queue ← queue ∪ mutated
        observations ← observations ∪ observation
    end if
end while
```

parameterized by functions:

- **initSeedCorpus**: Initialize a new seed corpus.
- **isDone**: Determine if the fuzzing should stop or not based on progress toward a goal, or a timeout.
- **choose**: Choose at least one candidate seed from the queue for mutation.
- **mutate**: From at least one seed and any observations made about the program so far, produce a new candidate seed.
- **eval**: Evaluate a seed on the program to produce an observation.
- **isInteresting**: Determine if the observations produced from an evaluation on a mutated seed indicate that the input should be preserved or not.

Figure 2.4: Fuzzing in a nutshell[19]

Another issue is that RTL models are not inherently executable. Hardware designs must, be simulated after being translated to a software model and being combined with a design-specific testbench and simulation engine, to form a Hardware Simulation binary (HSB). This means there needs to be a way for fuzzers to interact with HSBs that optimize performance and also trigger the HSB to crash upon detection of incorrect hardware behavior.

Simulating RTL hardware involves translating HDL into a functionally equivalent software (C/C++) model that can be compiled and executed. To accomplish this, most hardware simulators [link to verilator and iverilog] contain an RTL compiler to perform the translation. Verilator is a very popular open-source hardware simulator that translates SystemVerilog into a cycle-accurate C++ model for fuzzing. It is important to recognize that the way Verilator translates RTL hardware to software makes mapping software coverage to hardware coverage *implicit*.

2.4 Related Work

Significant work has been done in the field of RISC-V verification and coverage-guided fuzzing for RISC-V verification environments by the team at TU Bremen[22, 23, 24, 25]. V. Herdt et al. in their work in[23], have used coverage-guided fuzzing to verify three Instruction Set Simulator (ISS) models for RISC-V, of which one was a RISC-V simulator developed at TU Bremen[26]. The work was quite successful as it managed to uncover several new errors in the ISSs, and even an error in the official RISC-V reference simulator, Spike. The results of this fuzzer maximized most of the defined coverage metrics to 100% with the lowest metric at 81%. They suggest

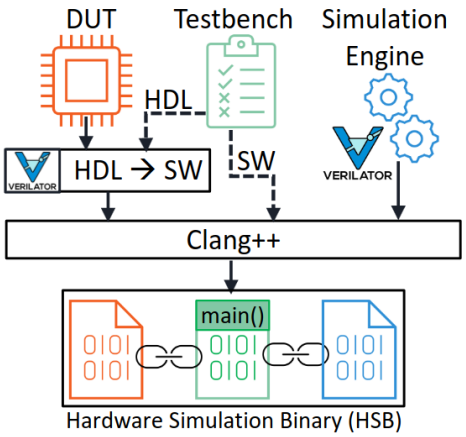


Figure 2.5: To simulate hardware, the DUT's RTL code is first translated to a software model, and then compiled and linked to a testbench and simulation engine to form an HSB. Executing this binary, simulates the behavior of the DUT[10].

implementing Constrained Random (CR) inputs as the seeds for the fuzzer to further improve performance, which will be taken into account in this thesis.

Similarly in [22], they have used Mutation-based approach to boost the quality of RISC-V Compliance tests[27]. Here, they have defined mutation classes and simulated it using RISC-V VP to improve coverage of the original RISC-V compliance tests. These papers, while very important to the field of mutation-based testing and in terms of their contribution of RISC-V, work mostly on an ISS rather than a RTL-based design, which as stated in Section 2.3.1 is a completely different form of implementation, in terms of fuzz testing.

Similar work has also been done by Schiavone et al. in [4], in which they have set up a completely open-source verification framework for a RISC-V core. The authors have used the evolutionary-generation program microGP(μGP) to generate test-cases based on the coverage metrics that was defined in the framework. μGP the generates instructions based on the coverage metrics that are to be fulfilled using an evolution-driven algorithm. The authors used ModelSim to run the simulation, and claim that the framework can also be achieved on any open-source simulator with sufficient SystemVerilog support, however this was not tested as part of this thesis. It is important to note that in [4] opted for a evolution-driven algorithm over coverage-guided fuzz testing for their framework, which require custom DUT-specific grammars to build assembly programs from.

More direct work has been done at the University of California, Berkeley, in regards to Coverage-Guided Fuzz Testing of RTL designs [9],[28]. In the paper [9], Lauefer et. al. have developed a tool that is called RFUZZ to implement coverage-guided fuzz testing for RTL designs on an FPGA. The first part of this tool is an instrumentation and harness generation component, which in theory can be used on any RTL design, regardless of the language it is written in as long as it can be described into FIRRTL[28]. This is quite a limitation because as of now, Chisel is the only language which can be translated to FIRRTL readily. Translating other HDLs, even one as widely used as SystemVerilog is experimental at best [10]. However, it could be a promising alternative, once the support for other languages in FIRRTL increases.

Trippel et. al. have published significant work on fuzzing hardware-like software. In this

work[10], they propose translating hardware designs to software models and fuzzing the model directly. Their approach has three key components:

- RTL hardware is translated to executable software
- Software fuzzers trace hardware coverage
- fuzzer-generated test cases are interpreted to effectively simulate the DUT

The first two steps can be handled by Verilator and fuzzer-provided compilers, the remaining component is given a more tailored approach. They create a bus-centric harness that can be a design- and fuzzer-agnostic verification framework, thereby demonstrating the versatility of adapting software testing tools for the hardware domain. This thesis focuses more on the processor-specific elements of the RISC-V ISA, and aims for a different set of coverage metrics to drive its fuzz engine.

3 Methodology

3.1 Overview of the Proposed Framework

As part of this thesis, a framework is proposed to provide a systematic approach to functional verification of a RISC-V processor by leveraging coverage-guided fuzzing. The methodology aims to maximize structural and functional coverage while detecting potential bugs in processor design. The methodology is also structured in a way that it can be modular and adaptable, allowing flexibility in implementation, regardless of the processor being tested.

The framework is built with an emphasis on:

- Functional Coverage: Ensure that the verification process covers all functional aspects of the DUT, including instruction execution, register operations, branching mechanisms, etc.
- Structural Coverage: Testing the DUT for errors in its implementation by covering internal states, control flow, transitions, etc.
- Flexibility and Adaptability: The framework should be modular and flexible enough to be rebuilt by others using different versions of the components and for different DUTs.

The framework is essentially a feedback loop that iteratively refines the testing process. It can be broadly divided into the following stages:

- Test Case Generation: Test cases are generated as a starting point to target various functional and structural aspects of the DUT, to provide meaningful coverage data. These can be generated to be completely random or guided by initial functional requirements.
- Simulation of tests: The generated test vectors are then executed on the DUT and a reference model, such as an ISA simulator. The results from both simulations are then logged for analysis.
- Coverage Analysis: The data logged from the simulation is analyzed and coverage metrics are updated to determine how thoroughly the processor's functionality are updated. The logs from both simulations are also cross-checked against each other to ensure that the results match.
- Feedback and fuzzing: Based on the coverage metrics that were updated, mutations are introduced in the test cases to improve coverage and focus on the gaps that were detected in the testing process.

Scalability, modularity, and reusability are the core principles involved in the design of this framework. The framework's modularity can allow users to, for example, replace the test generator with a custom one that is tailored to their DUT's requirements. The simulation environment can also be replaced with one of the user's choice given that the logs are made to be comparable. The coverage criteria are stored in a database-driven approach that makes it easy to add, remove, or modify any criteria.

3.2 Coverage-Guided Fuzzing Approach

Coverage-guided fuzzing is the heart of the proposed methodology, which allows a detailed exploration of the DUT and corner cases by making use of the feedback received from coverage analysis. The fundamental principle of coverage-guided fuzzing is to re-iteratively refine the test cases based on the updated coverage metrics.

First, the DUT needs to be instrumented to track the various coverage metrics. This includes making sure that the DUT is capable of logging the signals required for calculating coverage criteria that will be used to guide the fuzz engine. Depending on the design, this process may be as simple as just calling some pins in the testbench and logging their outputs, to manually defining the pins needed to map coverage within the DUT and using their outputs to calculate the coverage criteria.

The next step is to actually define the coverage criteria which can then be used to guide the fuzz engine. There are numerous options to choose from for coverage criteria, which can vary based on the DUT, the user, and the context of use. Some coverage metrics, like code coverage and branch coverage, can be easily obtained depending on the simulation environment. On the other hand more detailed metrics such as instruction coverage, register coverage, memory coverage etc. need to be defined by the user in a way that makes it easier to read and update during the simulation.

The coverage criteria along with the generated test corpus, form the inputs for the fuzz engine. With every iteration, the coverage data is updated which allows the fuzz engine to make changes to the test corpus to increase the coverage in the next iteration. The fuzz engine incorporates a number of heuristics to direct test generation towards meaningful exploration of the DUT. The goal of these heuristics is to ensure that fuzzing achieves both depth and diversity in coverage while avoiding redundancy. Some key heuristics that have been included as a part of the fuzz engine built for this thesis are:

- Dynamic Corpus Evolution: The test corpus evolves over iterations by incorporating low-coverage instructions and continuously refining existing test cases to target unexplored scenarios.
- Weighted Instruction Replacement: During each iteration, an some instructions from the test corpus are selected at random and replaced with instructions that have lower coverage. This selection process is guided by weights that are calculated from coverage data, ensuring that the instructions that are added to the test corpus do infact contribute to increasing the overall coverage.
- Feedback-Driven Refinement: After every simulation run, the current coverage state is updated. This allows the weights to be dynamically re-calibrated and avoid redundancies as more iterations are completed.
- Rare Instruction Targeting: Prioritize rarely executed or complex instructions to uncover edge cases.
- Boundary Memory Access: Test boundary conditions and unusual memory access scenarios, such as accesses near memory region edges or unaligned addresses.
- Arithmetic Edge Cases: Include operands designed to trigger corner cases like overflow, underflow, or division by zero.

Algorithm 1 Coverage-Guided Fuzzing

```
1: procedure CoverageGuidedFuzzing(T_0, ε)
2:     T ← T_0
3:     C ← ∅
4:     repeat
5:         for T_i ∈ T do
6:             ExecuteTest(T_i, DUT, ReferenceModel)
7:             δC ← ExtractCoverageData(ExecutionLogs)
8:             C ← C ∪ δC
9:         end for
10:        W ← CalculateWeights(C)
11:        U ← IdentifyUncoveredAreas(C)
12:        T_mut ← SelectTestCases(U, T)
13:        T_new ← MutateTestCases(T_mut, W)
14:        T ← T ∪ T_new
15:    until δC < ε
16:    T* ← T
17:    C* ← C
18:    return T*, C*
19: end procedure
```

Algorithm 1 provides more insight into the process of coverage-guided fuzzing. The process begins by initializing a test corpus T_0 and an empty coverage model C. Each test case T is executed on the DUT and the reference model, generating execution logs. These logs are analyzed to extract coverage data δC, which is then used to update the coverage model C. After processing all the instructions in the test corpus, weights W are calculated for each instruction based on the updated coverage model, with lower coverage instructions getting priority. Uncovered areas U are identified, and test cases relevant to these are areas are selected for mutation. The newly generated test cases T_{new} are merged with existing test corpus T, and the process is repeated. Iterations continue until the coverage improvement δC falls below a predefined threshold ϵ. Final outputs are the refined test corpus $T*$ and coverage model $C*$, representing a thorough exploration of the DUT's functionality.

3.3 Test Case Generation

The generation of the test corpus is a critical component of the framework, responsible for producing a diverse set of test cases that serve as the foundation for the coverage-guided fuzzing process. The test cases are generated programatically, balancing randomness with predefined constraints to ensure broad exploration of the instruction set and the DUT's functional scenarios.

The test generation process leverages a YAML file containing assembly instruction templates of the RISC-V ISA, which can be categorized by functionality. These templates also specify details like mnemonics, operands and constraints to ensure valid instruction generation. A user can easily add more instructions to this YAML file depending on which version of the ISA their DUT uses. Custom instructions can also be added easily in this manner.

The test generator allows for targeted test case generation by instruction category or for individual instructions for more granularity as well. The operands for each instruction are

generated dynamically which allows for better coverage with regards to criteria like register coverage, arithmetic edge cases and memory accesses. Once the test corpus is created, it is appended with 3 NOP instructions to the start and end and loaded into the simulation setup for execution. The NOP instruction makes it easier to spot the test instructions in the post-processing logs.

Algorithm 2 Generate and Validate Test Case

```
1: procedure GenerateAndValidateTestCase(C, M, N, mutate, coverage_data)
2:     T ← ["addi x0, x0, 0x0", "addi x0, x0, 0x0", "addi x0, x0, 0x0"]
3:     for i ∈ {1, 2, ..., N} do
4:         I ← GenerateInstruction(C, M, mutate, coverage_data)
5:         Template ← FetchTemplate(I.mnemonic)
6:         if ValidateInstruction(I, Template) then
7:             T ← T ∪ {I}
8:         else
9:             Log("Invalid instruction skipped")
10:        end if
11:    end for
12:    T ← T∪ ["addi x0, x0, 0x0", "addi x0, x0, 0x0", "addi x0, x0, 0x0"]
13:    WriteToFile(T)
14:    return T
15: end procedure
16: procedure GenerateInstruction(C, M, mutate, coverage_data)
17:    if M ≠ ∅ then
18:        Instructions ← {I | I ∈ Templates ∧ I.mnemonic = M}
19:    else
20:        Instructions ← {I | I ∈ Templates ∧ I.category = C}
21:    end if
22:    Template ← SelectRandom(Instructions)
23:    O ← ∅
24:    for op ∈ Template.operands do
25:        if op.type = "reg" then
26:            v ← WeightedRegister(coverage_data) if mutate else RandomRegister
27:        else if op.type ∈ ImmediateRanges then
28:            v ← WeightedImmediate(coverage_data) if mutate else RandomImmediate(op.type)
29:        end if
30:        O ← O ∪ {v}
31:    end for
32:    return FormattedInstruction(Template.mnemonic, O)
33: end procedure
```

Algorithm 2 outlines the approach to generating and validating the test cases. The process begins by initializing a test case (T) with a sequence of "no operation" (addi x0, x0, 0x0) instructions, which act as padding around the test code. For each test case, a new instruction is generated based on either its category or mnemonic, with optional mutation guided by coverage data to prioritize undertested areas. Each generated instruction is then validated against its

template to ensure correctness before being added to the test case (see lines 5 and 6). Invalid instructions are logged and skipped. Once all instructions are generated and validated, additional padding is added, and the test case is written to a file for execution. This method ensures the creation of robust and targeted test cases that enhance coverage, leveraging a combination of randomness and coverage-guided prioritization to uncover potential bugs effectively.

3.4 Test Execution and Monitoring

The next step in the methodology is to set up an execution environment in which the correctness of the DUT can be tested by comparing its behavior against a trusted reference simulator of the user's choice. It is important however, that the outputs generated by both simulators be comparable so that any failures in the DUT can be detected. This typically includes monitoring inputs and outputs of both simulators, tracking the starting and ending states of the simulators etc.

To facilitate reliable testing, the RISC-V core needs to be integrated into a simulation environment with the following components:

- DUT Integration: The DUT needs to be configured with the necessary inputs, including a memory array with the test corpus loaded into the program, and registers, clock and reset signals set to the same state every simulation.
- Reference Model: The reference model should be configured to be in the same initial state, and execute the same test cases for the verification process to be valid.
- Synchronized execution: Both the DUT and simulator are executed in a step-wise manner, while logging changes to important signals like the program counter, register values and memory states at each step. This should be fairly easy to configure with most modern simulators.

Proper monitoring is important for failure detection and tracing the point of execution when the bug occurred. For the DUT, all the logged signals can be found in the trace files such as the Value Change Dump (VCD) files. The log files for the reference simulator depends on the simulator itself. For the Spike ISS used in the thesis, the simulator provides a detailed log of each executed instruction and the changes made to register and memory in each step.

Another aspect of the framework that needs close monitoring is the coverage data. This is done by a post-processing script written to parse through the log files generated after simulation. Depending on how the simulation logs are formatted, multiple coverage criteria can be updated by parsing them. In this thesis, the simulation logs are formatted so that the post-processing script updates instruction, branch, register, immediate coverage and memory access patterns.

3.5 Bug Detection and Analysis

After setting up the monitoring for the simulators, it is now possible to actually detect any bugs that arise during the verification process. The primary objective is to uncover discrepancies in functional and architectural correctness, detect crashes or hangs, and then analyze the root cause of incorrect behavior.

The following techniques can be used for detecting bugs:

Algorithm 3 Detect and Analyze Bugs

```
1: procedure DetectAndAnalyzeBugs(LogDUT, LogSimulator, Timeout)
2:     for step ∈ ExecutionSteps do
3:         DUTState ← LogDUT[step]
4:         SimState ← LogSimulator[step]
5:         if DUTState ≠ SimState then
6:             Bug ← RecordDiscrepancy(step, DUTState, SimState)
7:             Bug ← ClassifyBug(Bug)
8:             LogBugDetails(Bug)
9:         end if
10:    end for
11:    if ExecutionTimeout(Timeout) then
12:        RecordBug("Timeout/Hang Detected", CurrentState)
13:    end if
14:    UpdateCoverageMetrics(LogDUT)
15:    return BugLogs
16: end procedure
17: procedure ClassifyBug(Bug)
18:    if Bug.PC_Mismatch then
19:        Bug.Type ← "Control Flow"
20:    else if Bug.RegisterMismatch then
21:        Bug.Type ← "Functional"
22:    else if Bug.MemoryMismatch then
23:        Bug.Type ← "Memory Access"
24:    else if Bug.Timeout then
25:        Bug.Type ← "Crash/Hang"
26:    else
27:        Bug.Type ← "Edge Case"
28:    end if
29:    return Bug
30: end procedure
```

1. Output comparison: Logs from the DUT and simulator are compared at an instruction-by-instruction level during test case execution to find any mismatches in program counter values, register updates, and memory transactions (depending upon the instructions executed).

2. Crash and Hang Detection: Unexpected traps or exceptions cause the simulation to crash and log an error output along with the instruction which caused said trap. In case the DUT goes into a non-responsive state (such as deadlocks or infinite loops), the DUT has a timeout exception which causes the stalled test to terminate.

3. Functional Mismatch: Any discrepancies between the outputs of the DUT and the reference model are logged as potential bugs. These can occur as incorrect arithmetic or logical calculations, divergence in branching behavior or memory alignment violations.

4. Edge Case Failures: Some special test cases which focus on edge conditions are given more priority by the fuzz engine. If these test cases cause any discrepancies they are counted as edge case failures (registers at maximum and minimum values, out of range immediate values etc).

The detected bugs are then grouped together into categories such as functional bugs, control flow bugs, memory access bugs, crash and hang bugs, and edge case bugs. Each bug is logged along with the line and clock cycle where the bug occurred. This makes it easier to pinpoint the bug on a waveform viewer such as GTKWave and find the root cause of the error. Algorithm 3 shows how the verification framework performs bug detection and collection. Once the fix is applied, the erroneous test case is run again a few times with variations to ensure that the fix did indeed work.

3.6 Toolchain and Environment

As mentioned in 1.3 one of the main objectives of this thesis is to implement the framework using only open-source software. The RISC-V core selected as the DUT for the thesis is the *PicoRV32*[29], which is defined as a "size-optimized RISC-V CPU core". It implements the RV32IMC instruction set but can be configured as RV32E, RV32I, RV32IC, or RV32IMC. While the CPU is meant to be used as an auxiliary processor in FPGA designs and ASICs, due to its smaller size and high f_{max}, it has been chosen as the DUT in this thesis for the easy to read source-code and pre-built support for verilator[30].

Verilator is an open-sourced simulator and claim to be the fastest Verilog/SystemVerilog simulator. Verilator is invoked with parameters similar to GCC or Synopsys' VCS. It "verilates" the specified Verilog or SystemVerilog code by reading it, performing lint checks, and optionally inserting assertion checks and coverage-analysis points. It outputs single or multi-threaded .cpp and .h files as "verilated" code. Verilator does not directly translate Verilog Hardware Description Language (HDL) to C++ or SystemC but rather compiles the code into a much faster and optimized model, which is then wrapped in a C++/SystemC module. The results are that a compiled Verilog model that executes even on a single thread is about 10x faster than standalone SystemC[30].

The *Spike RISC-V ISS* is chosen to act as the GRM. Spike has already been ratified as the official reference simulator by the RISC-V organization, and is the reason why it was chosen as the GRM[31]. The Spike simulator also provides a lot of flexibility in terms of logs, debugging

and also customizing the size and starting address of the memory map, which is a very helpful feature to have when testing for memory alignment.

GTKWave is an analysis tool used to perform debugging on Verilog or VHDL simulation models. It is not meant to be run interactively with the simulation, but relies on a post-mortem approach through the use of dump files[32].

All the scripts used in the thesis were written using *Python 3.10.12*. This includes the script used for test corpus generation, the fuzz engine, the scripts used to parse simulation logs, update coverage data and handle bug detection.

Finally, the entire thesis was run on an personal computer with an AMD®Ryzen 5 5600X 6-core (12-thread) processor and 32 GB of RAM, with the Ubuntu 22.04.4 LTS operating system.

4 Implementation

This chapter provides a detailed walkthrough of how the proposed framework for RISC-V processor verification was developed and optimized. Building a robust framework capable of thoroughly testing a RISC-V core required a systematic approach to integrate tools, manage data, and optimize performance. The decisions taken to reach the end stage of this thesis and the rationale behind them will be explained in this chapter in greater detail.

The framework was developed to address three primary tasks. First is to generate meaningful test cases capable of exploring the full functionality of the RISC-V core. Second, to simulate these test cases across both the DUT and a reference model to validate functional correctness. Third, it had to track coverage, detect bugs, and provide feedback to subsequent tests. It was important to strike a balance between achieving a good depth of exploring the DUT's state space without making the process too computationally cumbersome.

It is also important that the framework be modular, allowing different components to operate both independently and together equally effectively. This modularity also ensures that the framework can continuously evolve and be extended with new features when required.

The first step in the implementation process is to establish the foundation: setting up the tools and environment. The PicoRV32 core was chosen as the DUT for its readily available documentation, and for using minimal code that was not compatible with Verilator (unlike other more popular cores), Verilator was chosen for its performance as a cycle-accurate simulator, Spike for its reliability as a golden reference, and Python for its versatility.

4.1 Setup and Installation

As mentioned above, setting up the tools and environment was the first step in building the framework for testing and verification of the RISC-V core. This section describes the tools required, steps for installation, and any peculiar challenges that arose during this process.

4.1.1 RISC-V GNU Compiler Toolchain

The very first step that needs to be done is to set up a RISC-V GNU compiler toolchain, which is needed no matter which DUT or GRM the user chooses for their setup. This toolchain can be found on the RISC-V organization's Github repository[33]. It supports two build modes: a generic ELF/Newlib toolchain and a more sophisticated Linux-ELF/glibc toolchain. For the sake of this thesis, the first option was chosen as the test corpus was run on bare metal mode with its own prefix and suffix code that was provided along with the PicoRV32 source code. The toolchain itself had a large number of dependencies that needed to be installed which has been shown in the following code snippet (command shown for Ubuntu):

```
$ sudo apt-get install autoconf automake autotools-dev curl python3 python3-pip
    python3-tomli libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison
    flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev ninja-build
     git cmake libglib2.0-dev libslirp-dev
```

Listing 4.1: riscv-gnu-toolchain installation

The next step is to simply run the configure file, for which the installation path must be picked. At this point it is important to note that unless specified, the toolchain is set up for the RV64GC even on a 32-bit build environment. This will cause incompatibility while working with Spike down the line, so it was important to specifically build the toolchain for RV32IMC (which is the toolchain required for PicoRV32).

```
./configure --prefix=/opt/riscv --with-arch=rv32imac --with-abi=ilp32d
make -j$(nproc)
```

Listing 4.2: riscv-gnu-toolchain configuration

It is important to note that this takes a long time. There are also a variety of advanced options available for those that need it. However, those options were beyond the scope of this thesis and might be useful for other users. Detailed instructions for these options can be found on [33].

4.1.2 Spike and RISC-V Proxy Kernel

Now, it is important to note that a version of Spike and Proxy Kernel (PK) comes pre-installed with the RV32 Compiler toolchain. However, this version of the installation was not sufficient for the framework, as the configurations needed to be slightly altered to be able to create log files and also adjust the architecture of the pk which was installed.

Spike is the official RISC-V ISS that has been adopted by the RISC-V Foundation[31]. It supports almost every extension that has been ratified by the RISC-V foundation. For Spike, it is important to manually enable the commitlog option which allows Spike to create a log file after simulation. While this is not mentioned anywhere officially, it was clarified by the owners of the Spike repository on enquiry.

```
$ apt-get install device-tree-compiler libboost-regex-dev libboost-system-dev
$ mkdir build
$ cd build
$ ../configure --prefix=$RISCV --enable-commitlog
$ make -j$(nproc)
$ [sudo] make install
```

Listing 4.3: Spike installation

The RISC-V Proxy Kernel, pk, is a lightweight application execution environment that can host statically-linked RISC-V ELF binaries. It is designed to support tethered RISC-V implementations with limited I/O capability and thus handles I/O-related system calls by proxying them to a host computer. This package also contains the Berkeley Boot Loader, bbl, which is a supervisor execution environment for tethered RISC-V systems. It is designed to host the RISC-V Linux port[34].

Here too, the default installation steps build a 64-bit version of pk and bbl which is only compatible with RV64 toolchain. Therefore, it is important to build the proxy kernel for the 32-bit toolchain as shown here:

```
$ mkdir build
$ cd build
$ ../configure --prefix=$RISCV --host=riscv32-unknown-elf --with-arch=
    rv32i_zicsr_zifencei
```

```
$ make -j$(nproc)
$ make install
```

Listing 4.4: riscv-pk installation

In both Spike and pk, $RISC-V is the path under which riscv-gnu-toolchain was installed.

4.1.3 Verilator

Although Verilator can be installed using the Ubuntu packet manager, it does not install the latest version of Verilator. So, it had to be installed manually. The installation of verilator is pretty straightforward. One needs to simply follow the instructions as shown in the installation guide.

```
# Prerequisites:
sudo apt-get install git help2man perl python3 make autoconf g++ flex bison
    ccache
sudo apt-get install libgoogle-perftools-dev numactl perl-doc
sudo apt-get install libfl2  # Ubuntu only (ignore if gives error)
sudo apt-get install libfl-dev  # Ubuntu only (ignore if gives error)
sudo apt-get install zlibc zlib1g zlib1g-dev  # Ubuntu only (ignore if gives
    error)

git clone https://github.com/verilator/verilator   # Only first time

# Every time you need to build:
unset VERILATOR_ROOT  # For bash
cd verilator
git pull         # Make sure git repository is up-to-date
git tag          # See what versions exist
#git checkout master      # Use development branch (e.g. recent bug fixes)
#git checkout stable      # Use most recent stable release
#git checkout v{version}  # Switch to specified release version

autoconf         # Create ./configure script
./configure      # Configure and create Makefile
make -j `nproc`  # Build Verilator itself
sudo make install
```

Listing 4.5: Verilator Installation

4.1.4 PicoRV32

The PicoRV32 core was simply cloned from the repository[29]. The only pre-requisites for running the core was the riscv-gnu-toolchain and Verilator for simulation. However, on attempting to run the example testbench using Verilator, this error shows up.

```
Vpicorv32___024root__DepSet_h733510b4__0.cpp: In function 'void
    Vpicorv32___024root___sequent__TOP__1(Vpicorv32___024root*)':
Vpicorv32___024root__DepSet_h733510b4__0.cpp:1255:51: error: cannot convert 'std
    ::string' {aka 'std::__cxx11::basic_string<char>'} to 'QData' {aka 'long
    unsigned int'} in assignment
 1255 |      vlSelf->picorv32__DOT__new_ascii_instr = std::string("");
      |                                               ~~~~~~~~~~~
      |                                               |
      |                                               std::string {aka std::
    __cxx11::basic_string<char>}
In file included from Vpicorv32__ALL.cpp:8:
Vpicorv32___024root__DepSet_h733510b4__0__Slow.cpp: In function 'void
    Vpicorv32___024root___settle__TOP__2(Vpicorv32___024root*)':
Vpicorv32___024root__DepSet_h733510b4__0__Slow.cpp:235:51: error: cannot convert
     'std::string' {aka 'std::__cxx11::basic_string<char>'} to 'QData' {aka '
    long unsigned int'} in assignment
  235 |      vlSelf->picorv32__DOT__new_ascii_instr = std::string("");
      |                                               ~~~~~~~~~~~
      |                                               |
      |                                               std::string {aka std::
    __cxx11::basic_string<char>}
```

Listing 4.6: PicoRV32 Verilator bug

There seemed to be no acknowledgment of this error on the Yosys PicoRV32 git repository or anywhere else. However, after a lot of searching, this article on [35] was found where they had the same issues. In the blog, the author notes that the error is thrown by the code in PicoRV32 Verilog trying to assign an ASCII string to a 64-bit register. While this is an acceptable practice in Verilog and also works with iVerilog, it does not compile with Verilator. In the following code snippet, the culprit code is shown:

```
      reg [63:0] new_ascii_instr;
  `FORMAL_KEEP reg [63:0] dbg_ascii_instr;
  `FORMAL_KEEP reg [31:0] dbg_insn_imm;
  `FORMAL_KEEP reg [4:0] dbg_insn_rs1;
  `FORMAL_KEEP reg [4:0] dbg_insn_rs2;
  `FORMAL_KEEP reg [4:0] dbg_insn_rd;
  `FORMAL_KEEP reg [31:0] dbg_rs1val;
  `FORMAL_KEEP reg [31:0] dbg_rs2val;
  `FORMAL_KEEP reg dbg_rs1val_valid;
  `FORMAL_KEEP reg dbg_rs2val_valid;

  always @* begin
    new_ascii_instr = "";

    if (instr_lui)      new_ascii_instr = "lui";
    if (instr_auipc)    new_ascii_instr = "auipc";
    if (instr_jal)      new_ascii_instr = "jal";
    if (instr_jalr)     new_ascii_instr = "jalr";
    if (instr_beq)      new_ascii_instr = "beq";
    if (instr_bne)      new_ascii_instr = "bne";
    if (instr_blt)      new_ascii_instr = "blt";
    if (instr_bge)      new_ascii_instr = "bge";
    if (instr_bltu)     new_ascii_instr = "bltu";
    if (instr_bgeu)     new_ascii_instr = "bgeu";
    if (instr_lb)       new_ascii_instr = "lb";
    if (instr_lh)       new_ascii_instr = "lh";
```

```
27      if (instr_lw)        new_ascii_instr = "lw";
28      if (instr_lbu)       new_ascii_instr = "lbu";
29      if (instr_lhu)       new_ascii_instr = "lhu";
30      if (instr_sb)        new_ascii_instr = "sb";
31      if (instr_sh)        new_ascii_instr = "sh";
32      if (instr_sw)        new_ascii_instr = "sw";
33      if (instr_addi)      new_ascii_instr = "addi";
34      if (instr_slti)      new_ascii_instr = "slti";
35      if (instr_sltiu)     new_ascii_instr = "sltiu";
36      if (instr_xori)      new_ascii_instr = "xori";
37      if (instr_ori)       new_ascii_instr = "ori";
38      if (instr_andi)      new_ascii_instr = "andi";
39      if (instr_slli)      new_ascii_instr = "slli";
40      if (instr_srli)      new_ascii_instr = "srli";
41      if (instr_srai)      new_ascii_instr = "srai";
42      if (instr_add)       new_ascii_instr = "add";
43      if (instr_sub)       new_ascii_instr = "sub";
44      if (instr_sll)       new_ascii_instr = "sll";
45      if (instr_slt)       new_ascii_instr = "slt";
46      if (instr_sltu)      new_ascii_instr = "sltu";
47      if (instr_xor)       new_ascii_instr = "xor";
48      if (instr_srl)       new_ascii_instr = "srl";
49      if (instr_sra)       new_ascii_instr = "sra";
50      if (instr_or)        new_ascii_instr = "or";
51      if (instr_and)       new_ascii_instr = "and";
52      if (instr_rdcycle)   new_ascii_instr = "rdcycle";
53      if (instr_rdcycleh)  new_ascii_instr = "rdcycleh";
54      if (instr_rdinstr)   new_ascii_instr = "rdinstr";
55      if (instr_rdinstrh)  new_ascii_instr = "rdinstrh";
56      if (instr_getq)      new_ascii_instr = "getq";
57      if (instr_setq)      new_ascii_instr = "setq";
58      if (instr_retirq)    new_ascii_instr = "retirq";
59      if (instr_maskirq)   new_ascii_instr = "maskirq";
60      if (instr_waitirq)   new_ascii_instr = "waitirq";
61      if (instr_timer)     new_ascii_instr = "timer";
62    end
```

Listing 4.7: The code tries to assign a string value to a 64-bit register

As a fix, the author wrote a Python function which converted the ASCII strings into a combination of bits which has the code look like this:

```
1       reg [63:0] new_ascii_instr;
2     `FORMAL_KEEP reg [63:0] dbg_ascii_instr;
3     `FORMAL_KEEP reg [31:0] dbg_insn_imm;
4     `FORMAL_KEEP reg [4:0] dbg_insn_rs1;
5     `FORMAL_KEEP reg [4:0] dbg_insn_rs2;
6     `FORMAL_KEEP reg [4:0] dbg_insn_rd;
7     `FORMAL_KEEP reg [31:0] dbg_rs1val;
8     `FORMAL_KEEP reg [31:0] dbg_rs2val;
9     `FORMAL_KEEP reg dbg_rs1val_valid;
10    `FORMAL_KEEP reg dbg_rs2val_valid;
11
12    always @* begin
13          new_ascii_instr = {8'd0};
14
15
16      if (instr_lui)      new_ascii_instr = {8'd108,8'd117,8'd105};
17      if (instr_auipc)    new_ascii_instr = {8'd97,8'd117,8'd105,8'd112,8'd99};
```

```
18      if (instr_jal)      new_ascii_instr = {8'd106,8'd97,8'd108};
19      if (instr_jalr)     new_ascii_instr = {8'd106,8'd97,8'd108,8'd114};
20      if (instr_beq)      new_ascii_instr = {8'd98,8'd101,8'd113};
21      if (instr_bne)      new_ascii_instr = {8'd98,8'd110,8'd101};
22      if (instr_blt)      new_ascii_instr = {8'd98,8'd108,8'd116};
23      if (instr_bge)      new_ascii_instr = {8'd98,8'd103,8'd101};
24      if (instr_bltu)     new_ascii_instr = {8'd98,8'd108,8'd116,8'd117};
25      if (instr_bgeu)     new_ascii_instr = {8'd98,8'd103,8'd101,8'd117};
26      if (instr_lb)       new_ascii_instr = {8'd108,8'd98};
27      if (instr_lh)       new_ascii_instr = {8'd108,8'd104};
28      if (instr_lw)       new_ascii_instr = {8'd108,8'd119};
29      if (instr_lbu)      new_ascii_instr = {8'd108,8'd98,8'd117};
30      if (instr_lhu)      new_ascii_instr = {8'd108,8'd104,8'd117};
31      if (instr_sb)       new_ascii_instr = {8'd115,8'd98};
32      if (instr_sh)       new_ascii_instr = {8'd115,8'd104};
33      if (instr_sw)       new_ascii_instr = {8'd115,8'd119};
34      if (instr_addi)     new_ascii_instr = {8'd97,8'd100,8'd100,8'd105};
35      if (instr_slti)     new_ascii_instr = {8'd115,8'd108,8'd116,8'd105};
36      if (instr_sltiu)    new_ascii_instr = {8'd115,8'd108,8'd116,8'd105,8'd117};
37      if (instr_xori)     new_ascii_instr = {8'd120,8'd111,8'd114,8'd105};
38      if (instr_ori)      new_ascii_instr = {8'd111,8'd114,8'd105};
39      if (instr_andi)     new_ascii_instr = {8'd97,8'd110,8'd100,8'd105};
40      if (instr_slli)     new_ascii_instr = {8'd115,8'd108,8'd108,8'd105};
41      if (instr_srli)     new_ascii_instr = {8'd115,8'd114,8'd108,8'd105};
42      if (instr_srai)     new_ascii_instr = {8'd115,8'd114,8'd97,8'd105};
43      if (instr_add)      new_ascii_instr = {8'd97,8'd100,8'd100};
44      if (instr_sub)      new_ascii_instr = {8'd115,8'd117,8'd98};
45      if (instr_sll)      new_ascii_instr = {8'd115,8'd108,8'd108};
46      if (instr_slt)      new_ascii_instr = {8'd115,8'd108,8'd116};
47      if (instr_sltu)     new_ascii_instr = {8'd115,8'd108,8'd116,8'd117};
48      if (instr_xor)      new_ascii_instr = {8'd120,8'd111,8'd114};
49      if (instr_srl)      new_ascii_instr = {8'd115,8'd114,8'd108};
50      if (instr_sra)      new_ascii_instr = {8'd115,8'd114,8'd97};
51      if (instr_or)       new_ascii_instr = {8'd111,8'd114};
52      if (instr_and)      new_ascii_instr = {8'd97,8'd110,8'd100};
53      if (instr_rdcycle)  new_ascii_instr = {8'd114,8'd100,8'd99,8'd121,8'd99,8'
        d108,8'd101};
54      if (instr_rdcycleh) new_ascii_instr = {8'd114,8'd100,8'd99,8'd121,8'd99,8'
        d108,8'd101,8'd104};
55      if (instr_rdinstr)  new_ascii_instr = {8'd114,8'd100,8'd105,8'd110,8'd115,8'
        d116,8'd114};
56      if (instr_rdinstrh) new_ascii_instr = {8'd114,8'd100,8'd105,8'd110,8'd115,8'
        d116,8'd114,8'd104};
57      if (instr_getq)     new_ascii_instr = {8'd103,8'd101,8'd116,8'd113};
58      if (instr_setq)     new_ascii_instr = {8'd115,8'd101,8'd116,8'd113};
59      if (instr_retirq)   new_ascii_instr = {8'd114,8'd101,8'd116,8'd105,8'd114,8'
        d113};
60      if (instr_maskirq)  new_ascii_instr = {8'd109,8'd97,8'd115,8'd107,8'd105,8'
        d114,8'd113};
61      if (instr_waitirq)  new_ascii_instr = {8'd119,8'd97,8'd105,8'd116,8'd105,8'
        d114,8'd113};
62      if (instr_timer)    new_ascii_instr = {8'd116,8'd105,8'd109,8'd101,8'd114};
63    end
```

Listing 4.8: Fixing the bug by replacing the string with binary ASCII values

The importance of finding this blog cannot be overstated towards the completion of this thesis. With no official support about this problem that should be appearing for everyone, this singular

blog is the only solution for people who are not familiar with the internal workings of Verilator. This is a glaring example of how modern processor verification is a difficult topic to breach for novices, and how important it is to have more open-source information readily available online.

4.1.5 Python

Installing Python is pretty straight-forward, and can be done either by Ubuntu's package manager or from the python website.

4.1.6 GTKWave

There is a chance that GTKWave comes pre-installed with Ubuntu, depending on the version (this was the case for my OS), if not it can be easily installed from the Ubuntu package manager using;

```
1    sudo apt-get install gtkwave
```

This concludes the setup necessary to get started with actually building the testing framework. At this point it is important to note that, one of the major reasons behind the selection of these particular tools was also compatibility with open-source tools. Many of the tools that claim to be open-source usually end up requiring some connection to closed-source tools like Cadence, Synopsys etc.

4.2 Architecture of the Framework

The verification framework was designed to provide a systematic approach to testing and verifying the r32 core, integrating test generation, simulation, coverage tracking, and bug detection into a cohesive workflow. The architecture put forward in this thesis, places more emphasis on scalability, modularity, and automation to ensure ease of debugging, adaptability, and efficiency.

4.2.1 Architectural Overview

The framework consists of four main components that work together to iteratively test the r32 core:

1. **Test Generation Module**: This module is responsible for creating assembly-level test cases based on instruction templates and feedback from coverage data. It also takes input from the fuzz engine to generate instructions that cover more untested sections of the space state.

2. **Simulation Environment**: This module executes test cases on two platforms:
   - **Design Under Test(DUT)**: The test corpus is loaded into the memory of the PicoRV32 core and simulated using Verilator.
   - **Spike**: The test corpus is simulated using Spike and this result is used as the GRM, and the DUT's output is compared against this output to check for functional correctness.

   It also performs the important task of generating detailed log files that will later be used for coverage analysis and bug detection.

3. **Coverage Analysis Module**: The coverage analysis module parses the log files generated by the DUT to compute various coverage metrics. Apart from calculating coverage metrics, it also dynamically updates the coverage model, which is then fed to the Fuzz engine.

4. **Fuzz Engine**: This is the heart of the framework. In every iteration, it takes the updated coverage model as an input and uses the test generation module to target under-tested areas of the design by prioritizing uncovered instructions, registers, and immediate ranges etc.

5. **Bug Detection Module**: Parses, and compares the logs from DUT and Spike to look for discrepancies, and then categorizes and logs them for debugging.

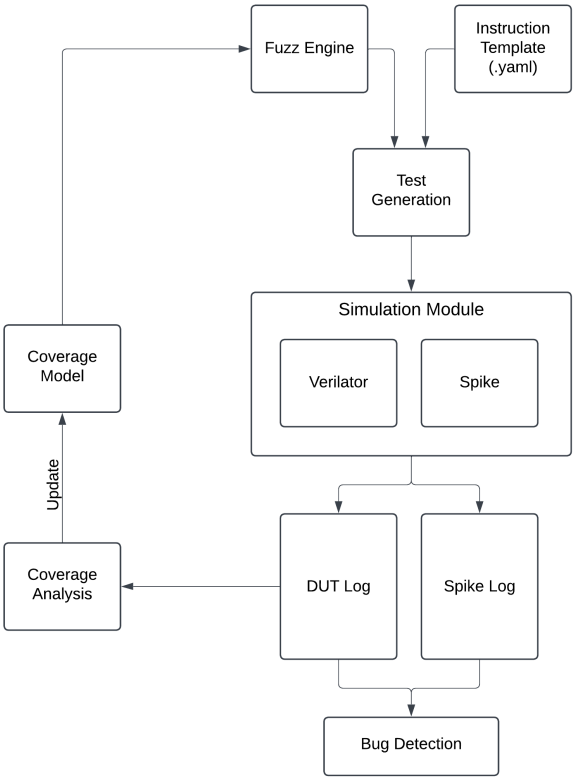


Figure 4.1: Architecture diagram

4.2.2 Test Generator

The test generator is the starting point of the framework. It creates test cases from instruction templates that are stored in a YAML-based template file. This file contains the syntax, operands, and constraints for all the supported instructions. Using these templates, the test generator was used create a seed corpus of test cases, which serve as the foundation for the fuzz engine to work on. This seed corpus also serves the purpose of achieving an initial coverage value for the metrics in the coverage model for the fuzz engine to read from.

4.2.3 Fuzz Engine

The **Fuzz Engine** enhances the generated test cases by introducing mutations, guided by coverage data from iterations. It operates as part of a feedback-driven loop. The fuzz engine applies targeted mutations to existing test cases in the seed corpus, based on the insights it receives from the Coverage Analysis module.

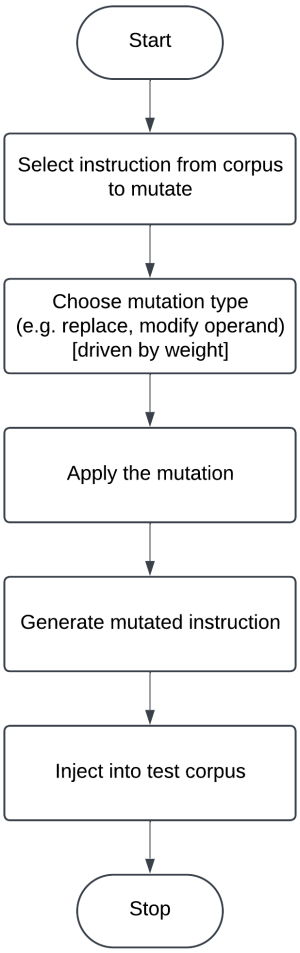


Figure 4.2: Simplified working of the flowchart

The fuzz engine uses a dynamic mechanism where weights are assigned to instructions and operands based on their coverage metrics. This mechanism prioritizes areas with lower coverage, which ensures sufficient exploration of the DUT's state space, while also avoiding redundant testing by reducing the chances of that mutation appearing again as its coverage passes a certain threshold.

4.3 Core Functionality

This section dives into the essential operations of the framework, and highlights the processes that makes the testing and verification environment efficient and effective. This includes the generation of test cases, their execution on the simulation environment, tracking and updating coverage metrics, and detecting bugs.

4.3.1 Test Case Generation

The Test case generator is the cornerstone of the framework, as it creates assembly level test cases tailored by the fuzz engine to be simulated and verified. The starting point for the test

generator is a YAML file that contains all RV32IMC instructions in a particular format that defines its name, operands, constraints, and a category that was defined just for the sake of this thesis and is not supported in any official capacity by the RISC-V foundation. The ADDI instruction for e.g., looks like this in the YAML file.

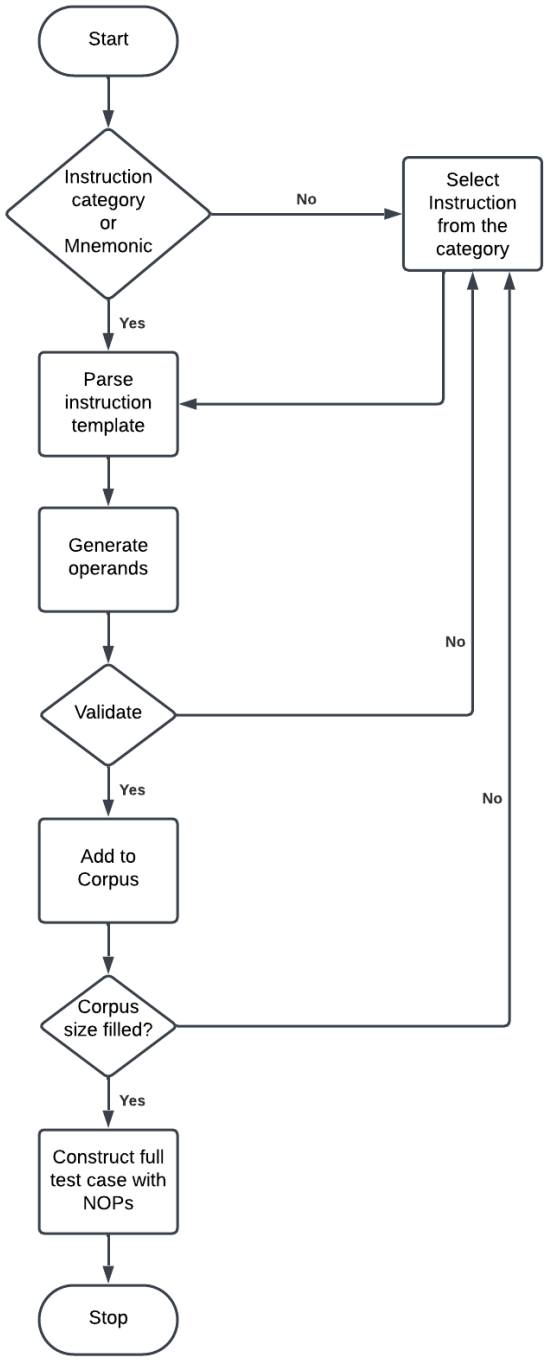
```
    ADDI:
      mnemonic: "ADDI"
      operands:
        - rd: "reg"
        - rs1: "reg"
        - imm: "signed_12bit"
      constraints:
        imm_range: "-2048 to 2047"
      description: "Add immediate"
      category: "arithmetic_logical"
```

Listing 4.9: Instruction template for the ADDI instruction

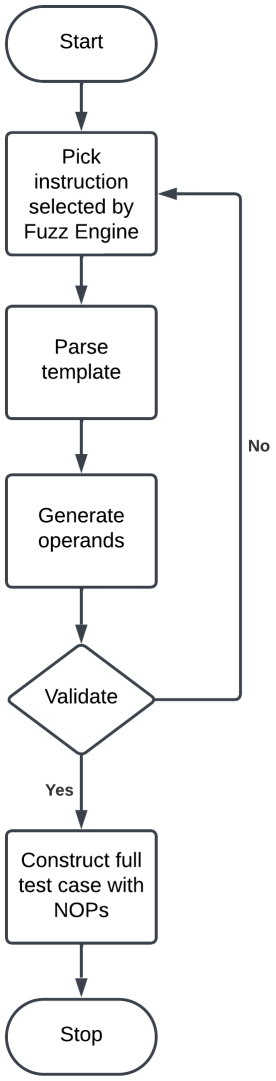
The test_generator.py script is responsible for generating these test cases in a structured and dynamic manner. It takes as input the template YAML file as well as the output from the fuzz engine. To create the initial seed corpus, the user can select to build it around a specific category or around specific instructions. This provides the benefit of having multiple smaller test vectors which can then be tested in parallel increase coverage in real time. This allows for parallel testing of seed corpuses that may not necessarily overlap with each other.

Once an instruction is selected from the template file, the register and immediate values are filled at random to generate a r32 assembly instruction. Once the instruction is generated, the script validates the generated instruction to make sure that there was no error in the generation of the instruction. In case of an error, the instruction is discarded and a new one is generated, otherwise it simply moves on to the next instruction. This process goes on until the desired number of test cases are generated, and then the now complete test corpus is padded with three No Operations (NOPs) instructions to the beginning and the end. This makes it much easier for the coverage analysis and bug detection modules to parse through the clunky log files generated by Verilator and Spike.

In the event that the test generation is not happening for the first time, then instead of selecting an instruction at random, a mutation is introduced by the fuzz engine. Based on what the proposed mutation was, the test generator, then parses the instruction and selects the instruction or operand that needs to be mutated and introduces a mutation that is a legal RV32IMC instruction. These mutated instructions are revalidated and if found proper, then replace the older instruction in the test corpus.



(a) Test Generation logic for the seed corpus



(b) Test Generation logic when it is crating mutation for the test corpus

Figure 4.3: Figure shows the functional logic of the test generator script when it is creating the seed corpus as well as when it is controlled by the fuzz engine

The final test corpus, in both cases, is saved to a .S file that can be compiled, built and linked to build two different .elf files which can be simulated by the DUT and the GRM.

4.3.2 Simulation Environment

The simulation environment forms a critical part of the framework, enabling the execution of generated test cases and providing a platform for validating the r32 core's functionality. It enables accurate and efficient comparison between the DUT and the GRM.

DUT Simulation with Verilator

The PicoRV32 core is the DUT and it came pre-loaded with a well-structured approach to run the test code. Their approach was to have single .S file that includes all the other test files. This .S file is then compiled into a .elf file that is then converted into a .hex file using a python script. This .hex file is finally loaded into the memory array that is defined in the Verilog testbench by the developers. All read and write operations are performed on this memory array and therefore can be easily tracked by the testbench.

The testbench did not however, create any useful log file that could be used for coverage analysis or bug detection. However, after looking through the simulations on GTKWave it was discovered that there were a lot of useful signals that were hidden at different hierarchical levels from the testbench but visible in the VCD file. To take advantage of this the Python library called vcd.reader was used to parse through the VCD file, recognize the useful signals and log their values at the end of every clock cycle. In the VCD file, each wire is represented by a token of a unique set of one or two characters that remain consistent. It is simply a matter of finding the token that pertains to the required signals and making a dictionary of them.

Once the log is created from the VCD file, the next step is to find the actual test code. Since the test code was bordered on both sides with 3 NOPs instructions, the large VCD-parsed log is then parsed again to take only those lines which lie between the triple NOPs.

Spike Simulator

Running the test code on Spike is much easier than running it on the DUT. The test corpus needs minimal changes to be done to it. Just some prefix code is needed to align the test code. After that, it can be compiled and run on Spike with the help of pk and the bbl loader. Spike provides the option of setting the simulation memory with a particular base address and particular size. Using this feature one can have a simulation memory for Spike that is identical to the memory used by the DUT. The way to set this up is quite self-explanatory from the Spike help menu, if one knows the memory map of the DUT.

Spike also has an option to output a log file after simulation which is very suitable for parsing through and is also perfectly suited for coverage analysis. Spike only logs the Program Counter (PC), the instruction executed, and those registers or memory location that were updated. From the Spike logs, same as before everything not within the triple NOPs are discarded.

The logs that are generated from the VCD files and Spike log are extremely large (in the range of hundreds of MiB) and quickly add up in terms of space consumed, especially when simulating so many instructions. Therefore it is important to discard the unwanted instructions as quickly as possible.

4.3.3 Coverage Analysis and Bug Detection

Once the reworked log files are ready from both simulations to be parsed for coverage and bug analysis. A single script was written to parse both scripts and perform the task of coverage analysis and bug detection.

Working of Coverage Analysis

The script integrates tightly with the simulation logs created in the previous stage. At the core of its design, are efficient parsing mechanisms that leverage regex-based pattern matching to extract relevant data from the log files. This includes program counters, instruction execution details, register values, CPU states and memory states.

Instruction coverage tracks how many times each opcode is executed. As instructions are parsed from the log, their execution counts are updated within the coverage model. This ensures that the coverage model dynamically reflects real-time execution data. **Register coverage** records read and write access patterns across all registers. Each register involved in an operation is tracked for both read and write counts to enable targeted fuzzing. **Immediate Coverage** focuses on validating immediate values against predefined ranges specified in the coverage model. For each immediate value parsed, the script evaluates whether it falls within a specific range, and updating hit counts and marking the ranges as covered. While it is pretty simple, **state machine coverage** is also an important part of coverage model. For the purposes of this thesis, the PicoRV32 core has a wire that tracks the CPU state, which is tracked by the script. It also helps that there are only 5 states to the core, of which only 3 states are important.

The program counters serve as synchronization anchors between the observed and expected states. By comparing program counter values, the script detects deviations that could signify control flow anomalies.

Since the coverage analysis script is run after every iteration, it means that the coverage model is not static and changes with every iteration. This provides a dynamic feedback loop between the coverage model and the fuzz engine, ensuring that the framework continuously refines its focus, targeting areas with lower coverage.

Comparison of Observed and Expected States

One of the most fundamental functionalities of the framework is to compare the outputs of the DUT and GRM simulation logs. The observed state is based on the log generated by Verilator and the expected state is based on the log generated by Spike. Discrepancies between these states are logged as potential bugs, which were categorized as:

- Control Flow Bugs, which are discrepancies in PC values, which indicate unintended jumps or stalls.
- Functional Bugs, which are discrepancies in the register values suggesting incorrect instruction execution or data flow.
- Memory Access bugs, which are discrepancies in memory reads or writes, linked to load/-store instruction errors.

The script finally produces two primary outputs:

- **Updated Coverage Model**: The coverage.yaml file is augmented with new execution counts, hit counts for immediate ranges and access patterns for registers. This updated model is fed back to the fuzz engine to guide the next iteration of test case generation.
- **Bug Logs**: A structured log of detecting bug is saved as a YAML file. Each entry contains details about the bug type, a program counter, instruction opcode, and the observed versus expected states.

The design of the coverage analysis module prioritizes precision, adaptability, and feedback integration. The reliance on dynamic parsing ensures that the framework can handle diverse test cases and large simulation logs without sacrificing accuracy.

4.3.4 Fuzz Engine

The fuzz engine is the central mechanism of the entire framework. Its purpose is to iteratively refine test cases, using feedback from the coverage analysis module to prioritize under-explored regions of the DUT. By incorporating weighted probabilistic selection, the fuzz engine provides an adaptive, yet systematic approach to functional verification.

Design and Workflow

The fuzz engine is seamlessly integrated with the test generator and coverage analysis modules. It employs the following workflow to generate and refine test cases:

1. **Seed Corpus Generation**: The first test corpus that is created without any input from the fuzz engine ends up forming the foundation for the subsequent test corpuses that undergo mutation.

2. **Weighted Mutation Process**: The core of fuzzing engine lies in the mutation mechanism, where instructions or their operands undergo mutation based on the status of the coverage model in that iteration of simulation. This includes, but is not limited to, register substitution, immediate adjustment, or even a completely different instruction.

3. **Iterative Refinement**: After the other blocks simulate the newly mutated instructions and the coverage model is updated, the fuzz engine reapplies the mutation process to target other areas of lower coverage. This iterative loop continues until the desired coverage is met or the maximum number of iterations is reached.

Weighted Probabilistic Selection

If the coverage model were divided into just two categories of **reached coverage threshold** and **did not reach coverage threshold**, it would cause the coverage-guided fuzzing to be not very different from random verification. This would mean that a coverage criterion that is nearly at threshold and a coverage criterion that has a value of zero both have an equal chance of being mutated into the test corpus the next iteration. This slows down the convergence rate to high coverage significantly.

Instead we assign a weight to all coverage elements which biases the fuzz engine towards exploring parts of the DUT's state space that have lower coverage, while still maintaining a level of randomness. For an element e, the weight $w(e)$ is calculated as:

$$w(e) = \frac{C_{max} + S}{C(e) + S} \tag{4.1}$$

where:
C_{max}: Maximum target for coverage normalization
$C(e)$: Current coverage count of the element
S: Smoothing factor to prevent excessively large weight disparities

4.4 Challenges and Solutions

A challenge that presented early in the process, while setting up the simulation environment was how to access the data from the internal signals that were necessary to calculate coverage data as well as cycle accurate machine state. The easy and non-elegant solution would be to force the internal signals as outputs by changing the source codes of the PicoRV32 core and then calling them in the testbench as monitoring pins. However, this would be a very specific fix that may not apply past this thesis. On searching for a more elegant and modular solution, the Python vcd.reader library was discovered. This was instantly adopted as the way to go because even the most basic Verilog simulator provides the ability to generate a .vcd file. This way the solution proposed in the thesis would be processor agnostic.

Another challenge that presented itself once the simulations started running were that logs were too large and after a few simulations, the author's PC started running out of space. Additionally, it was difficult to sort out the actual test code from the mountain of prefix and suffix code for both simulations. As a fix to both, in the thesis, the test code was bordered with 3 NOP instructions on both sides, making it easy to pick out from the large sea of instructions and consequently dropping the unwanted data from the log files making them significantly smaller.

Mutated instructions would sometimes violate the RV32 ISA which led to failed simulations. This happened many times, and many hours into the verification process before the author realized that the failure was due to an illegal instruction and not a bug. This prompted the addition of a validation function in the fuzz engine script.

4.5 Optimization and Performance Improvements

As the work on the thesis continues, some obvious areas for improvement showed up. Here we document these optimizations that were made to the process:

- Initially, there was not weighted calculations for introducing mutations. This meant that, although, there was dynamic coverage updates, there was a very slow convergence rate towards high coverage, because it was essentially performing random verification. Once this problem was diagnosed, weighted mutation selection was introduced which led to faster convergence rates towards full coverage.

- Parallel execution was implemented owing to the nature of the author's PC. Multiple seed corpuses were generated with minimal interdependency, i.e., a different corpus for testing branch coverage, a different corpus for testing memory accesses etc. This meant the only overlapping coverage criteria would be register, immediate and some memory access coverage. Having these test corpuses running in parallel meant the overlapping criteria reached the convergent point faster and also the different types of instructions were also tested more robustly because they could have a test template that matched their function.

5 Results and Discussion

In this chapter, the results of all the experiments conducted using the setup that was mentioned in the previous sections will be presented. Multiple simulation runs were conducted with different parameters to different results and this section contains the most interesting ones.

For the sake of having a final coverage goal, each coverage criteria is given a number of times it is covered. This number directly corresponds to the value C_{max} from Eqn.(4.1). 90% was set as the threshold value for the coverage. The number of instructions in each iteration of the test was set to 100, with a mutation percentage of 10, that is, every iteration 10% of the tests undergoes mutation. The smoothing factor S for calculating the weights from Eq n.(4.1) was set to 5. The limit of iterations was set as 200, because earlier testing showed that most criteria reach full coverage within 200 iterations (some branch-related coverage were exceptions to this rule). Finally, each iteration was made to run 4 test corpuses in parallel, so technically, one iteration counts as four.

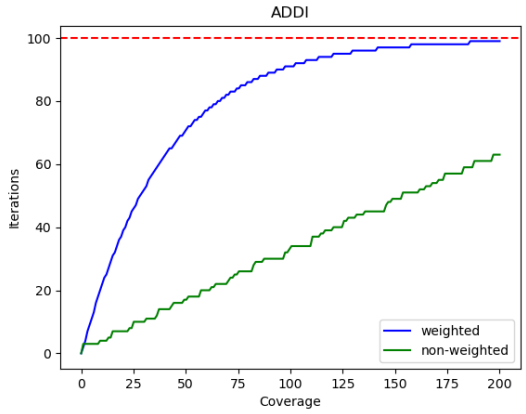
5.1 Results

The time taken to run 200 iterations, on an average comes to around 4317 seconds. The same, tests when run in parallel on 4 cores, bring down the average time to 1241 seconds. This results in a roughly 70% speed-up in verification, which is seen throughout the whole process. The total runtime to achieve the coverage as shown in 5.1 comes to 10951 seconds or 11.37 hours.

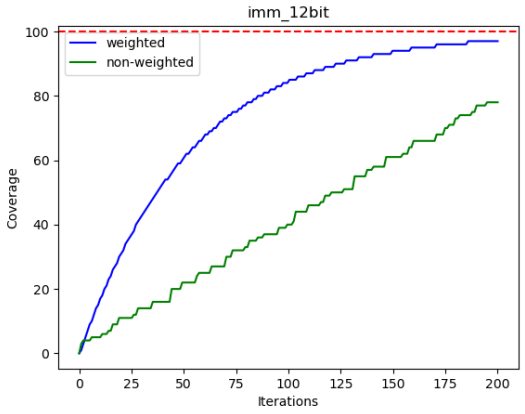
Criteria	Coverage	# Iterations	Time taken(s)
Instruction	100.0	2271	9872
Register	94.72	1562	6882
Immediate	91.36	2533	11003
Branch	81.0	3000	13176
FSM	100.0	1	18
Total runtime			40951

Table 5.1: Coverage achieved by the verification framework and the time taken for the simulations

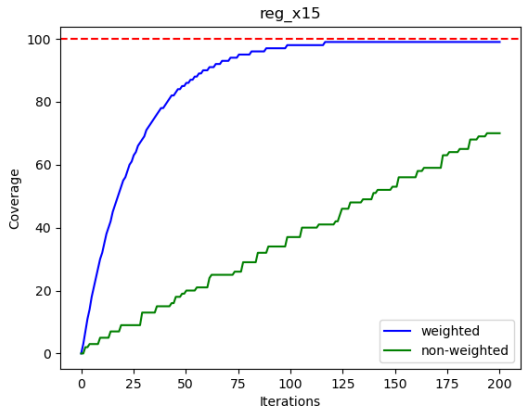
As mentioned in the optimizations section, the verification environment started out without the weighted mutation approach. The results of that were quite interesting. See Fig. 5.1, and here we quantify the difference made.



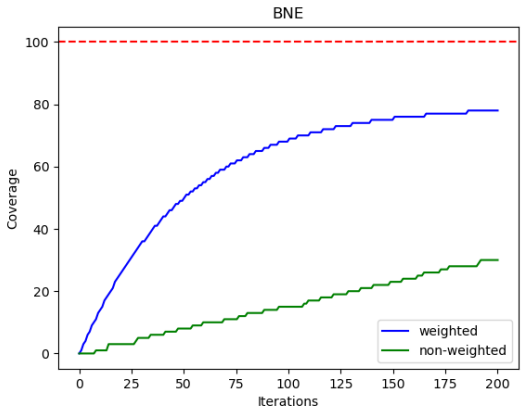
(a) ADDI: with vs without weights



(b) imm_12bit: with vs without weights



(c) reg_x15: with vs without weights

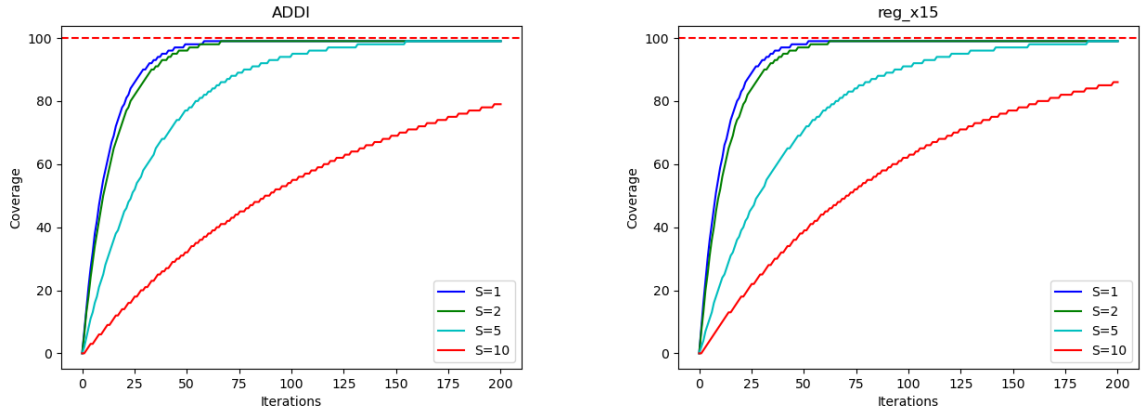


(d) BNE: with vs without weights

Figure 5.1: Figure shows the difference in number of iterations taken to achieve full coverage (in most cases) with weighted mutation selection vs without. Tests were set up particularly to make this comparison and were allowed to run the entire 200 iterations, with $S = 5$.

The difference made by introducing weighted selection is clearly visible in 5.1. In 5.1a we can see that at the end of 200 iterations, the unweighted run has only reached a coverage of about 60%, which the weighted run finished in about 75 minutes, which makes it roughly 67% faster. Similarly for 5.1b, at the end of 200 iterations, the unweighted run has only reached a coverage of 80, making the weighted run 25% faster. Similarly, we can observe that in 5.1c the weighted run is 60-70% faster. In 5.1d, neither run reaches full coverage, but the weighted run is still over a 100% faster than the unweighted. This makes clear the benefit of controlling the mutation with some logic on top of the coverage-driven guidance that was initially proposed.

Another factor that was experimented with was S, the smoothing factor from (4.1). Mathematically, a lower smoothing factor would create an extreme bias towards low coverage instructions and operands, but what would be its effect on verification efficacy?



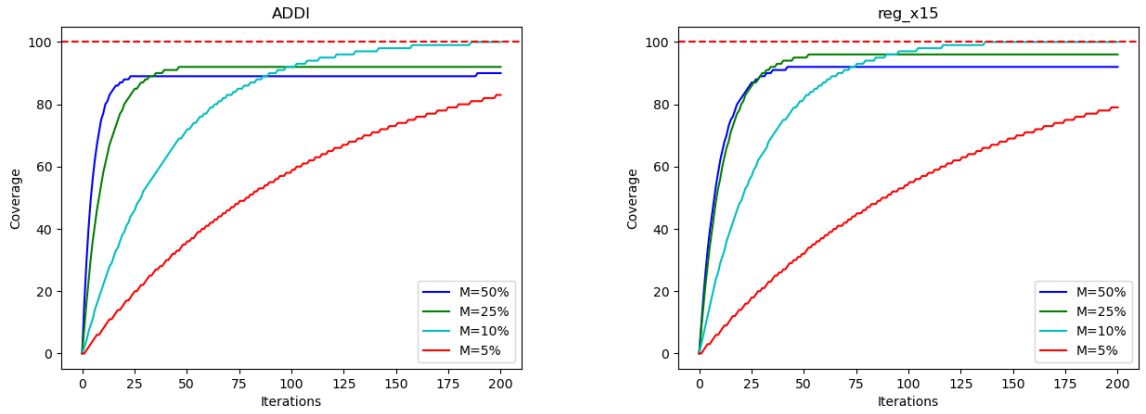
(a) ADDI: with 4 different values of S

(b) reg_x15: with 4 different values of S

Figure 5.2: In this figure two different coverage criteria have been simulated 4 times each. Every time with a different smoothing factor.

The benefit of having a lower smoothing factor is that the coverage criterion reaches the acceptable threshold of 90% coverage extremely quickly (see S=1 in 5.2a and 5.2b). However, the drawback is that at such low S-values, the weight of coverage criteria for the remaining 10% of its state space is practically zero, which could be leaving some important edge case scenarios untested.

Until now all mutations were introduced at a rate of 10%. This meant that every iteration 10% of the test corpus was selected to be replaced with the mutated instructions. In this section, the mutation percentages(M%) is varied across a range of 5% to 50%.



(a) ADDI: with 4 different values of M%

(b) reg_x15: with 4 different values of M%

Figure 5.3: In this figure two different coverage criteria have been simulated 4 times each. Every time with a different M%s.

From the graphs, especially 5.3b it seems as if as the M% increases, the number of iterations needed to reach the target coverage falls by quite a bit. However, when 50% of the test corpus is replaced every time there is a loss of stability, the new mutations need not necessarily contribute to the coverage, and it becomes hard to keep track of important edge cases that may break the design. Looking at the graphs of 5.3 and 5.2, makes the author think what would happen if

there was a deliberate attempt to control both factors.

While conducing this experiment was beyond the scope of this thesis, one can hypothesize that since increasing M% controls how many instructions are altered each iteration, which increases the diversity of test cases. While on its own, this would just be chaotic because there would be random instructions, if one were to combine the increase M% with a reduced S, it would potentially fill the increased "spaces" with instructions that have lower coverage count. One could also imagine a dynamic approach to this fine-tuning of M% and S, which would speed-up the exploration of the DUT's state space in earlier iterations and later can be inverted to keep testing without jeopardizing the stability of the system.

5.2 Analysis of Detected Bugs

The PicoRV32 core is a very robust core that is contributed to regularly, by around 30 professional computer scientists for close to a decade. So, there is little chance of finding an actual bug in the processor using this verification framework. Instead, for the purpose of this thesis, faults will be injected into the source code, to create a mismatch between DUT and Spike logs. The injected faults and their respective status in the bug log is given in 5.2.

Fault Injected	How it was handled
Flip bits in decoder	Logged as either control flow or functional bugs (depending on which instructionwas triggered)
Swap values written to RD with another register	Mismatch in register values are logged as functional bugs
Set the branch prediction logic to be always taken	Logged as control flow error
Introduce wrong state transitions	Logged as memory access or functional bugs depending on which state was triggered
Flip bits of immediate values	Either results in unexpected operand usage or functional error

Table 5.2: Table shows what faults were injected into the core and how they were handled by the verification framework

6 Conclusion and Future Work

6.1 Conclusion

In summary, a verification framework for RISC-V cores based on coverage-guided fuzzing was built using only open-source software. The goal of this framework was to improve the effectiveness and efficiency of hardware verification process. The core of the verification framework was the use of coverage-guided fuzzing, which increased the effectiveness of the the verification progress by targeting untested parts of the processor. By utilizing the open-source simulator Verilator, the framework also required much less computational time and power, making it a more efficient process.

However, the framework is limited by what kind of cores it can verify, as well as which test generators it can use or not use, makes the modular design a bit redundant for the current time. Hopefully, in the future, when the RVVI is more popular, the framework can be reworked to be truly modular and design-agnostic.

Despite these limitations, the results of this work demonstrate the potential of coverage-guided fuzz testing in RISC-V even using purely open-source material.

6.2 Future Work

One of the first steps towards improving this framework, would be to make it more design-agnostic. This would allow the framework to be tested on more processors in the real world. This would also require the framework to be scaled up to handle multi-threaded, and pipelined processors and other more complex designs.

Introducing more advanced mutation strategies that would help increase coverage with reduced runtime. Machine learning techniques can also be incorporated to guide the mutation and test generation process.

Finally, expanding the coverage model to include micro-architectural features like branch prediction units, pipeline stages, and cache coherence mechanisms would provide deeper insights into the DUT's behaviour while ensuring comprehensive verification.

Bibliography

[1] "Tech Reports — EECS at UC Berkeley — www2.eecs.berkeley.edu." https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html. [Accessed 11-10-2024].

[2] "RISC-V Members — riscv.org." https://riscv.org/members/. [Accessed 11-10-2024].

[3] "NVIDIA Shipping Around One Billion RISC-V Cores In Their 2024 Products — phoronix.com." https://www.phoronix.com/news/RISC-V-NVIDIA-One-Billion. [Accessed 11-10-2024].

[4] P. D. Schiavone, E. Sanchez, A. Ruospo, F. Minervini, F. Zaruba, G. Haugou, and L. Benini, "An Open-Source Verification Framework for Open-Source Cores: A RISC-V Case Study," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 43–48, Oct. 2018. ISSN: 2324-8440.

[5] C. Wolf, "End-to-end formal isa verification of risc-v processors with riscv-formal," in *7th RISC-V Workshop Proceedings*, 2017.

[6] "GitHub - riscv-verification/RVVI: RISC-V Verification Interface — github.com." https://github.com/riscv-verification/RVVI. [Accessed 11-10-2024].

[7] M.-R. et al., "A compact functional verification flow for a risc-v 32i based core," in *2020 IEEE 3rd Conference on PhD Research in Microelectronics and Electronics in Latin America (PRIME-LA)*, pp. 1–4, 2020.

[8] T. Liu, R. Ho, and U. Jonnalagadda, "Open Source RISC-V Processor Verification Platform,"

[9] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: coverage-directed fuzz testing of RTL on FPGAs," in *Proceedings of the International Conference on Computer-Aided Design*, (San Diego California), pp. 1–8, ACM, Nov. 2018.

[10] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing Hardware Like Software,"

[11] "About RISC-V International — riscv.org." https://riscv.org/about/. [Accessed 11-10-2024].

[12] D. A. Patterson and A. Waterman, *The RISC-V reader : an open architecture atlas*. Berkeley, California: Strawberry Canyon LLC, first edition (book version) 1.0.0. ed., 2017.

[13] I. Newsroom, "X86: approaching 40 and still going strong — telecomtv.com." https://www.telecomtv.com/content/industry-announcements/x86-approaching-40-and-still-going-strong-27183/. [Accessed 11-10-2024].

[14] A. Waterman, K. Asanovic, and C. Division, "The RISC-V Instruction Set Manual: Volume 1,"

[15] S. Gal-On and M. Levy, "eembc.org." https://www.eembc.org/techlit/articles/coremark-whitepaper.pdf. [Accessed 12-01-2025].

[16] S. Vasudevan, *Effective functional verification: principles and processes*. Dordrecht: Springer, 2006. OCLC: 70268008.

[17] M. Barnasconi, F. PÃªcheux, and T. VÃ¶rtler, "Advancing system-level verification using UVM in SystemC,"

[18] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: a survey," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 4, p. 123â193, Apr. 1999.

[19] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, (Toronto Canada), pp. 2123–2138, ACM, Oct. 2018.

[20] A. Fioraldi, A. Mantovani, D. Maier, and D. Balzarotti, "Dissecting american fuzzy lop: A fuzzbench evaluation," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, Mar. 2023.

[21] R. Kannavara and S. Larson, "Fuzzing soft ips for fun & profit,"

[22] V. Herdt, S. Tempel, D. GroÃe, and R. Drechsler, "Mutation-based Compliance Testing for RISC-V," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, (Tokyo Japan), pp. 55–60, ACM, Jan. 2021.

[23] V. Herdt, D. GroÃe, H. M. Le, and R. Drechsler, "Verifying Instruction Set Simulators using Coverage-guided Fuzzing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 360–365, Mar. 2019. ISSN: 1558-1101.

[24] N. Bruns, V. Herdt, and R. Drechsler, "Processor Verification using Symbolic Execution: A RISC-V Case-Study," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, (Antwerp, Belgium), pp. 1–6, IEEE, Apr. 2023.

[25] N. Bruns, V. Herdt, and R. Drechsler, "Unified HW/SW Coverage: A Novel Metric to Boost Coverage-guided Fuzzing for Virtual Prototype based HW/SW Co-Verification," in *2022 Forum on Specification & Design Languages (FDL)*, pp. 1–8, Sept. 2022. ISSN: 1636-9874.

[26] V. Herdt, G. o. C. Architecture, and R. Drechsler, "RISC-V VP," Jan. 2025. original-date: 2018-06-19T14:09:48Z.

[27] A. B. et. al, "riscv-compliance/doc/README.adoc at master Â· lowRISC/riscv-compliance — github.com." https://github.com/lowRISC/riscv-compliance/blob/master/doc/README.adoc. [Accessed 11-10-2024].

[28] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 209–216, 2017.

[29] "GitHub - YosysHQ/picorv32: PicoRV32 - A Size-Optimized RISC-V CPU — github.com." https://github.com/YosysHQ/picorv32?tab=readme-ov-file#building-a-pure-rv32i-toolchain. [Accessed 11-10-2024].

[30] W. Snyder, "Veripool — veripool.org." https://www.veripool.org/verilator/. [Accessed 11-10-2024].

[31] "GitHub - riscv-software-src/riscv-isa-sim: Spike, a RISC-V ISA Simulator — github.com." https://github.com/riscv-software-src/riscv-isa-sim. [Accessed 11-10-2024].

[32] "GTKWave — gtkwave.sourceforge.net." https://gtkwave.sourceforge.net/. [Accessed 11-10-2024].

[33] "GitHub - riscv-collab/riscv-gnu-toolchain: GNU toolchain for RISC-V, including GCC — github.com." https://github.com/riscv-collab/riscv-gnu-toolchain. [Accessed 12-01-2025].

[34] "GitHub - riscv-software-src/riscv-pk: RISC-V Proxy Kernel — github.com." https://github.com/riscv-software-src/riscv-pk. [Accessed 12-01-2025].

[35] "Analyze PicoRV32 - HackMD — 30vhEV7FQECcWeCF1eAN5A." https://hackmd.io/@30vhEV7FQECcWeCF1eAN5A/S1ybboVnK. [Accessed 12-01-2025].