UC Santa Cruz
UC Santa Cruz Electronic Theses and Dissertations

Title

Improving Effectiveness and Productivity of Microprocessor Verification

Permalink
https://escholarship.org/uc/item/83t7t9hk

Author
Kabylkas, Nursultan Nuridenuly

Publication Date
2022

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

IMPROVING EFFECTIVENESS AND PRODUCTIVITY OF MICROPROCESSOR VERIFICATION

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Nursultan Nuridenuly Kabylkas

December 2022

The Dissertation of
Nursultan Nuridenuly Kabylkas is approved:

Professor Jose Renau, Chair

Professor Martine Schlag

Professor Heiner Litz

Peter Biehl
Vice Provost and Dean of Graduate Studies

Copyright © by

Nursultan Nuridenuly Kabylkas

2022

Table of Contents

List of Figures **vi**

List of Tables **viii**

Abstract **ix**

Acknowledgments **xi**

1 Introduction **1**
- 1.1 Microprocessor Verification Challenges . . . 1
- 1.2 Contribution of the Dissertation . . . 3

2 Background **7**
- 2.1 Typical Verification Setup . . . 7
- 2.2 Random Instruction Generators . . . 8
- 2.3 Reference Model Comparison . . . 9
  - 2.3.1 End-of-simulation comparison . . . 10
  - 2.3.2 Trace comparison . . . 10
  - 2.3.3 Co-simulation . . . 11
- 2.4 Formal Verification . . . 11

3 Single core co-simulation **13**
- 3.1 Introduction . . . 13
- 3.2 Dromajo . . . 14
- 3.3 Checkpoint Definition . . . 14
- 3.4 Verification Flow with Dromajo . . . 15
  - 3.4.1 Checkpoint Generation . . . 16
  - 3.4.2 Step and Compare . . . 16
- 3.5 Dromajo Integration . . . 17
- 3.6 Deterministic RISC-V co-simulation . . . 19
- 3.7 Related Work . . . 20

4 Marionette Models: going beyond single-core co-simulation 22
4.1 Introduction 22
4.2 Marionette Models 25
4.3 Marionette Model of CVA6 Branch Predictor 27
4.3.1 Step 1: Marionette Skeleton Design 27
4.3.2 Step 2: Marionette Joint Design 27
4.3.3 Step 3: Marionette Building 28
4.3.4 Step 4: String Hooking 28
4.3.5 Step 5: Staging the Marionette Play 29
4.4 Marionette Model Integration Details 29
4.5 Why Co-simulation Does Not Work for Multi-core? 30
4.5.1 Inapplicability of Co-simulation on a Shared-Memory Multi-Core System 30
4.5.2 Incapability of the Co-simulation to check for Memory Ordering Specification 33
4.6 Building Shared Memory System's Marionette 35
4.6.1 Step 1: Marionette Skeleton Design 35
4.6.2 Step 2: Marionette Joint Design 36
4.6.3 Step 3: Marionette Building 39
4.6.4 Step 4: String Hooking 39
4.6.5 Step 5: Staging the Marionette Play 40

5 Logic Fuzzer: enhancing the state-of-the-art 43
5.1 Introduction 43
5.2 Logic Fuzzer 45
5.2.1 Congestors: A Case for Fuzzing 45
5.2.2 Table Mutatators 47
5.2.3 Stressing mispredicted path 48
5.3 Can the LF's states ever happen in the real world? 51
5.4 Logic Fuzzer Implementation 51
5.5 Related Works 53
5.5.1 Input-stimuli fuzzing 53
5.5.2 Fault Injection 54

6 Evaluation 55
6.1 Evaluation Methodology 55
6.1.1 RISC-V Cores 55
6.1.2 Evaluation Metrics 57
6.1.3 Test Binaries 57
6.2 Evaluation 58
6.2.1 Main Results 58
6.2.2 Dromajo Evaluation 58
6.2.3 Dromajo+Logic Fuzzer Evaluation 62

6.2.4 Dromajo+Marionette Models Evaluation 64
6.2.5 Dromajo+Marionette Model+Logic Fuzzer Evaluation 66
6.3 Discussion 69
6.3.1 Bug Hunting with LF and False Positives 69
6.3.2 Toggle Coverage 70
6.3.3 Marionette Model Integration 71

7 Conclusion and Future Opportunities 73
7.1 Conclusion 73
7.2 Future Work 74

Bibliography 77

List of Figures

3.1 Verification flow with Dromajo 16
3.2 RTL-Dromajo interaction flow 17

4.1 Simulation test-bench. Illustration of how the hardware components ”pulls the strings” of the Marionette Model. 29
4.2 Execution threads of memory operations. 31
4.3 Timing diagram of a multi-core system executing memory operations (x-axis is time). (a) to (d) emphasize the events occurring at the highlighted stages. 32
4.4 Execution threads of memory operations with intermediate buffering. . . 35
4.5 The “skeleton” of the shared memory system’s marionette. 37
4.6 Pipeline timing diagram of a multi-core system displaying activities happening at clock cycle 11 41

5.1 Congestor Logic Fuzzer placed at the FIFO’s full signal 46
5.2 CVA6’s L1 cache way/bank utilization without (a) tag array mutation and (b)(c) with tag array mutation 47
5.3 The coverage of instructions that were in CVA6’s mispredicted path . . 49
5.4 Instruction address ranges retrieved from Branch Target Buffer while executing random instructions. 50
5.5 RTL-Fuzzer interaction flow 51

6.1 Number of branch misprediction of 16 different branch predictor Marionette Models along with the branch predictor of CVA6. The simulation is running fraction of the Linux Boot. The Marionette Model with the same BHT size as CVA6’s branch predictor behaves identically. 65
6.2 Timing diagram of memory operations going through the dual-core BOOM pipeline and pulling strings of the Marionette. 66
6.3 Zoomed-in timing diagram with annotations of memory operations going through the dual-core BOOM pipeline. 68
6.4 The dynamics of unique event ordering occurrence during the simulation with and without Logic Fuzzer. 69

6.5 Coverage increase when running verification binaries 71

List of Tables

4.1 List of API functions and the corresponding conditions under which they are called. For brevity purposes, the signal names were shortened. All of these signals are defined in CVA6's frontend.sv. 28
4.2 Simple test code 31
4.3 Pipeline stages of a simplified out-of-order processor. 32
4.4 Litmus test 34
4.5 List of API functions and the corresponding conditions under which they are called. For brevity purposes, the signal names were shortened. . . . 40

6.1 Summary of the cores used for evaluation 57
6.2 Summary of the simulated tests 57
6.3 Summary of the bugs exposed by Dromajo 59
6.4 Summary of the bugs exposed in three RISC-V cores by Logic Fuzzer enhanced Dromajo. 62

Abstract

Improving Effectiveness and Productivity of Microprocessor Verification

by

Nursultan Nuridenuly Kabylkas

The study on verification trends in the semiconductor industry shows that the design complexity is increasing, fewer companies achieve first silicon success, companies hire more verification engineers, and 53% of the whole hardware-design-cycle is spent on the design verification [17]. The cost of a respin is high, and more than 40% of the cases that contribute to it are post-fabrication functional bug exposures [15]. The study also shows that 65% of verification engineers' time is spent on debugging, test creation, and simulation [16].

In this dissertation, I discuss tools and methodology that help verification engineers to expose more bugs before fabrication and increase the productivity of time spent on design debugging, test creation, and simulation. In particular, first, I discuss Dromajo, the state-of-the-art processor verification framework for RISC-V cores. Dromajo is an RV64GC emulator that was designed specifically for co-simulation purposes. It can boot Linux, handle external stimuli, such as interrupts and debug requests on the fly, and can be integrated into existing testbench infrastructure with minimal effort.

Second, I address a major limitation of co-simulation as a technique. Previously, it had been impossible to co-simulate multi-core processor configurations. In this

dissertation, I talk about Marionette Models, a methodology that, for the first time ever, enabled the co-simulation of microprocessor designs in the multi-core settings.

Finally, I discuss Logic Fuzzer (LF), a novel tool that expands the verification space exploration without the creation of additional verification tests. The LF randomizes the states or control signals of the design-under-test at the places that do not affect functionality. It brings the processor execution outside its normal flow to increase the number of microarchitectural states exercised by the tests.

I evaluate the effectiveness of the tools on three RISC-V cores: CVA6 [54], BlackParrot [3], and BOOM [56]. The tools and methodology described in this dissertation expose bugs in the designs and enhance the state-of-the-art verification techniques.

Acknowledgments

A dissertation is an enormous task, but fortunately, it was not done alone. This dissertation would not be possible without God's blessing and, by His will, the direct and indirect contributions of various individuals I met along the way.

First and foremost, I would like to express my deepest gratitude to my advisor Jose Renau who has been a true mentor and a passionate teacher throughout my time at UCSC. Professor Renau has been my role model whom I have always been looking up to. His brilliance, creativity, deep understanding of processor architecture, being an exceptional systems programmer and software engineer, and many other characteristics have always been constant sources of awe. I thank Professor Renau for believing in me, for being patient with me during my personal hardships, and for supporting me throughout my graduate school career.

I would also like to thank the reading committee members, Dr. Martine Schlag and Dr. Heiner Litz. Thank you for your early support and valuable feedback during my advancement. Special thanks to Dr. Litz for the support during the MICRO conference and attending my talk.

I thank all of the Professors at UCSC with whom I had a chance to interact. I thank Professor Matthew Guthaus for his exceptional VLSI courses. I thank Professor Ricardo Sanfelice for his unique cyber-physical systems course, giving me positive feedback and making me believe in myself during the first year of grad school. I thank Professor Patrick Tantalo for teaching the Algorithms course that gave me the right

mindset when writing software. I thank professor Suresh Lodha for the data visualization course that gave me the right mindset when communicating data through plots. I thank Professor Manfred Warmuth for giving me fundamental knowledge in Machine Learning. I thank Professor Jose Renau once again for the insightful courses on Computer Architecture. Thanks to Professor Sagnik Nath, Professor Anujan Varma and once again to Professor Jose Renau for allowing me to TA their classes.

I thank all of my lab mates at the MASC research group at UCSC - Sheng, Ramesh, Akash, and Nilufar. I appreciate all your help and support. Special thanks to my senior lab colleagues Rafael and Daphne. Thanks for addressing all of my novice questions during the first year at MASC lab.

During my Ph.D. journey, I was fortunate to do internships which contributed to my professional development immensely. One of those internships was in an amazing start-up Esperanto Technologies (ET). I am grateful to Dave Ditzel for forming this company and giving me the opportunity to work with true experts. I am grateful to the many brilliant individuals I met at ET. Thanks to Tommy Thorn, Chronis Xekalakis, and Shreesha Srinath. Thanks for all your advice, feedback, and insightful conversations we had during lunches. This dissertation wouldn't be possible without your input. In fact, it is more proper to say that this dissertation is an input to your project. Thanks to Mike Neilly for trusting me and allowing me to contribute to Esperanto's awesome performance simulator. Thanks to the management of the company who decided to open-source their RISC-V emulator which I heavily used in my research. My special thanks go to Jayesh Iyer. Jayesh, I extremely appreciate all your support both in

professional and personal matters. Thank you for being a true mentor and friend. Thank you for the countless hours you spent explaining the architecture of ET's chip and giving me insights about computer architecture in general.

Another internship I want to emphasize was at a company that needs no introduction – Advanced Micro Devices. I am tremendously lucky for ending up in a team with world-class engineers, true professionals, and simply amazing people, Amit Ben-Moshe and Justin Smith. Under Amit's guidance, I learned the difference between the hacky dirty code I had been writing and the quality production software which I aim to continually learn to produce at AMD. Amit, thank you for being a true mentor. Thank you for your guidance, valuable feedback, and great textbook suggestions. Justin, thank you for teaching me about AMD GPU architecture. Thank you for answering all my ignorant questions in a manner that not only provides clarity but also provides a vast load of insight. In my opinion, your email responses can be compiled into a sophisticated textbook. Finally, I would like to thank Timour Tursunovich Paltashev. Timour Tursunovich, thank you for believing in me and for making impossible things possible, for example, my current affiliation with AMD. I am fascinated with your sense of social responsibility and constant readiness to give and help. Thank you for all the activities you conduct within and outside the AMD. Thank you for all your help and support in professional and personal matters. I thank all my AMD colleagues for their patience and for waiting for me to complete this dissertation for such a long time.

I also would like to thank two people without whom my existence would not be possible, my dear dad Nuriden Yerzhanov and my dear mom Tattak Yerzhanova. Thank

you for literally dedicating your lives to our upbringing. Thank you for your infinite love, and constant support and for making us who we are today. I thank my elder brother Arman for always supporting me in whatever I do. Thank you for handling all the physical house chores when I had to prepare for the CS olympiads and high school exams.

I am saving my biggest gratitude to my wife and three precious kids without whom I would have finished this dissertation three years ago. Jokes aside, I would like to thank, from the bottom of my heart, – my wife Arai. Janka, you were my inspiration, my motivation, and the only cheerleader throughout grad school. I wish UCSC could put your name on the diploma besides mine since we went through this together. Thanks to our three precious kids Hamza, Sarah and Shakira. Being near them, smelling and kissing them was one of the few working antidotes against the stressful moments I had.

Last but not least I thank myself for all my hard work and dedication.

Chapter 1

Introduction

> I am not free and independent; I am
> a traveler with duties.
>
> Said Nursi

1.1 Microprocessor Verification Challenges

Modern microprocessors are complex systems. The study on verification trends in the semiconductor industry shows that the design complexity is increasing, and fewer companies achieve first silicon success, and companies hire more verification engineers, and 53% of the whole hardware-design-cycle is spent on the design verification [17].

When verifying microprocessors, the common practice is to build a *co-simulation* infrastructure [31, 32]. The co-simulation compares the execution of the design-under-test (DUT) against the execution of a high-level software model of the processor, also known as the "golden" model. The underlying idea is simple: when we run the code on

the DUT and the model, the architectural state must be the same at any moment. In the case of a mismatch, reasons are investigated, and bugs are uncovered.

To get confidence about DUT correctness, engineers use various proxy metrics, such as automatic code-based and circuit-structure-based coverage, as well as manual implementation-specific functional coverage [45]. Another typical metric is to track the bugs found per week. The verification team thoroughly studies the architecture specification and carefully designs the functional coverage models. The plan must cover ample verification space and, most importantly, consider the corner cases [19]. The team then sets the goals for the metrics mentioned above. The DUT is intensely stressed by adding numerous tests to regression, gradually uncovering bugs, and increasing the coverage until defined goals are achieved.

In the context of the described state-of-the-art verification practices, I want to bring up three specific challenges.

First, co-simulation as a technique has fundamental issues that limit its capabilities (challenge #1). To be specific, co-simulation cannot be applied to multi-core systems. Also, co-simulation cannot check the correctness of the processor's performance-enhancing systems such as branch predictor.

Second, the described infrastructure does not guarantee that the processor is bug-free when sending the design for fabrication (challenge #2). The study on verification trends shows that fewer companies achieve first silicon success due to increased complexity and need costly re-spins before production. Despite the immense amount of co-simulation and achieved high coverage, 40% of the reasons that cause a re-spin

are functional bugs that escape to silicon [17]. These bugs get exposed only during the silicon validation or, even worse, at the end customer. The verification engineers call these bugs *outlier bugs* or *simulation resistant super bugs*. The outlier bugs are exposed neither by random instruction streams nor by directed tests because the sequence of events for the bug to occur is too complicated. They can only "be exposed by exercising the design outside its normal flow or operating parameters" [5].

The third challenge is that verification requires large amounts of human effort and resources (challenge #3). It is reported that 53% of the whole hardware-design-cycle is spent on design verification. Out of that time, to reach the defined coverage goals, verification engineers spend 21% of their time generating tests and running them on the simulator. When bugs are uncovered, they spend 44% of their time debugging [17].

1.2 Contribution of the Dissertation

This dissertation makes an effort to address the issues mentioned above. First, to be better than the state-of-the-art, we should start at the state-of-the-art level. Hence, we built the platform for our experimental studies. The artifact of this dissertation is the-state-of-the-art co-simulation framework for RISC-V cores. We called it Dromajo [46], and it is open-source released. Dromajo is an RV64GC emulator that was designed specifically for co-simulation purposes. It can boot Linux, handle external stimuli, such as interrupts and debug requests on the fly, and be integrated into existing testbench infrastructure with minimal effort. Dromajo addresses the need for

productivity and shortens the debug-verify cycle. The co-simulation framework simplifies debugging because an engineer starts the investigation at the point closest to the divergence of the model and the DUT. Our results show that the mere integration of Dromajo into the testbenches of existing open-source RISC-V cores exposed bugs and proved their effectiveness. Besides being, Dromajo has some features that, to the best of our knowledge, are unique and not supported by any existing infrastructures. These features improve the productivity of the verification engineers addressing challenge #3. Read more about Dromajo in Chapter 3.

Next, I address the fundamental limitations of the state-of-the-art (challenge #1). The abstraction level of the reference model is too high to verify many critical components of the processor. The current verification infrastructures models a single-cycle machine that operates on the instruction level granularity, primarily to verify the functional correctness at the ISA level. This setup makes it impossible to verify processor components that span their logic across several pipeline stages or even go beyond the core into caches. Specifically, the reference model completely abstracts away a branch predictor despite its importance to the performance. Similarly, the reference model completely removes the intricacies of memory request orderings involved in shared memory multi-core systems. One way to solve these issues is to build software that models the complexity mentioned above. However, this approach nearly replicates the original RTL implementation, which makes it impractical or, at the least, a high-cost solution.

We propose a practical and inexpensive solution, Marionette Models (MM).

These models are utterly oblivious to the hardware complexity, and updates/accesses to the micro-architectural state happen only when notified by the hardware. We draw an analogy with a marionette puppet where the complex behavior, such as human body movement, is reduced to pulling strings. The marionette is a simple software model that mimics the human. A human is a complex non-deterministic system that acquires the knowledge of when to pull the specific string. The strings are the set of API functions that connect two systems. Read more about MM in Chapter 4.

Finally, I present a methodology to address challenge #2, Logic Fuzzer. It is a technique that brings the processor's execution outside of its normal flow and increases the chances of finding outlier bugs in the simulation phase. The key in simulation-based verification is to develop the code sequence that will bring the processor into a buggy microarchitectural state that results in an inconsistent architectural state with the golden model. The Logic Fuzzer increases the microarchitectural states reached *without* the engineering effort of developing new code sequences, hence increasing the productivity of time spent on the test creation. The idea is to randomize states or control signals in the DUT that do not affect the correctness. For example, the re-order buffer (ROB) may assert a full or stall signal even when it is not full. It is also possible to change the branch predictor tables at any given moment or even insert the instructions in the mispredicted path. The Logic Fuzzer can change the number of cycles but does not corrupt the functionality or the program order. Our results show that atypical microarchitectural states created by Logic Fuzzer expose more bugs during the simulation phase.

Logic Fuzzer is not to be confused with the fuzzing of input stimuli [28, 35, 42]. It is the fuzzing of the actual logic. The inserted logic stirs up the execution paths while running the code and brings the processor outside its normal flow. It does not require specialized code and operates independently of existing verification infrastructure.

In this dissertation, I demonstrate that the presented tools and methodology can expose hardware malfunctions and inconsistencies with ISA that could prevent any complex software from running correctly. I also discuss interesting observations related to Operating Systems. Three RISC-V cores that we used for evaluation have gone through several tape-outs and claim to boot and run Linux. More than half of the bugs found were OS related. Interestingly, a "well behaved" Linux will not have exercised most of the bugs. Our results show that being able to boot and run Linux is far from saying that the core is verified.

The rest of the dissertation is organized as follows. Chapter 2 provides background concepts. Chapter 3 goes over Dromajo and its main contributions. Chapter 4 describes Marionette Models. In Chapter 6, I describe different types of Logic Fuzzer and their implementation details. Finally, I finish with concluding remarks and future work in Chapter 7.

Chapter 2

Background

> New occasions in life are like adding new binaries to a regression-testing. You exercise parts of yourself that hasn't been touched previously. Manifestation of bugs is expected and normal. So don't panic. Take time to fix the bugs, and move on as an improved version of yourself.
>
> — Nursultan Kabylkas

2.1 Typical Verification Setup

In simulation-based verification, the RTL model of the microprocessor is translated into a high-level object-oriented software class [44]. The translated high-level

model is then instantiated in the testbench along with the memory model. The memory gets prepopulated with the verification tests. After the simulation infrastructure is set up, we must come up with criteria to determine whether the tests that we are simulating pass or fail. The trivial option is the correctness checking by running directed tests. When a test completes execution, the final result is checked against a pre-calculated answer. The test's success or failure is determined based on the comparison of the test output and the pre-calculated answer. The directed tests are heavily used in industry, but often the purpose of these tests is to verify the basic functionality of the design or, on the contrary, to reach a well-thought-out corner case. For more than three decades, to test the design at a more rigorous level, designers both in industry and academia relied on the verification binaries that are randomly generated.

2.2 Random Instruction Generators

Random Instruction Generator, also known as Test Program Generator or Instruction Stream Generator, is a utility software that generates randomized assembly instruction streams given the set of configurations. The tests generated by the RIG sweeps a broad range of implemented functionality. It can create complex test cases that are hard to come up with for an engineer [2, 52].

Stressing the design-under-test with complete random instruction streams can be thought of as testing the system in "breadth." Complete randomness does not provide control over the generated tests. The probability of hitting a bug that is hidden "deep

down” under the complex interactions among different units is very low. To close this gap, some RIGs give the ability to have control over the generated tests through test program templates. The template is an abstract description of the test and describes a set of constraints that the generator should satisfy. Hence, it is giving the ability to manage the direction and the “depth” of the generated tests [9, 49].

The next question is: how do we determine if the verification code failed or passed? Self-checking techniques are not applicable due to the random nature of the generated tests. We can solve this issue by building an infrastructure that compares the execution with the reference model.

2.3 Reference Model Comparison

The reference, or *the golden* model, is the high-level software model of a processor. The characteristics of such a model are that it is fast and uncomplicated, it does not reflect any details of the implementation and the changes to the architectural state happen in instruction-level granularity.

The reference model comparison is the verification technique that, as the name suggests, compares the execution paths of the implementation and the model. The underlying idea is simple: when we run the code on the DUT and the model, the architectural state of both must be the same at any given moment. We can implement the comparison with the reference model in different ways with different levels of complexity.

2.3.1 End-of-simulation comparison

The end-of-simulation comparison is a cheap and simplistic setup. This method compares only the architectural state once the test completes. In other words, the same code is run both on the reference model and the RTL implementation. At the end of the simulation, we dump the register file states and the memory of both and compare against each other. If any of the values do not match, the reasons are investigated. This approach's drawback is that a buggy behavior that got reflected in the architectural state can be overwritten and hidden by later correct execution. Besides, even when the mismatch is detected, another problem is that it is challenging to debug as we might be very far from the point of divergence [26].

2.3.2 Trace comparison

Another way to implement a reference model checking setup is through trace comparison. This method needs both of the models to be able to dump the execution logs. Typically, these logs contain information about program counter flow and every single register/memory writeback. The traces are then compared, and mismatches are flagged. This approach solves both of the issues mentioned above.

Nevertheless, this technique fails to work when we test an external stimulus, such as interrupts and debug requests. Due to their asynchronous nature, an external stimulus can fire completely randomly during the execution. Because in the described infrastructure, both models are running standalone and comparisons happen post factum, the single interrupt will cause execution logs to be different [31].

2.3.3 Co-simulation

To tackle the issue described in Section 2.3.2, an infrastructure that runs both models in parallel and supports communication between the models must be built. The cores simultaneously start executing the same code and pass messages to one another. In general, we must support two types of messages.

First, at the specified event, for example, when an instruction is committed, the RTL model will signal the reference to commit an instruction and compare the states of interest. The failed comparison immediately halts the execution, and the stimulus that caused it is reported. This approach simplifies debugging because an engineer starts the investigation at the point closest to the divergence.

Second, to support asynchronous interrupts, the setup must support the messaging that overwrites the emulator's execution path. When the RTL flags an interrupt, it must be able to inform the emulator to follow its execution path [1].

2.4 Formal Verification

Formal verification techniques have shown promising results [25] and are getting more popular [8, 51]. This type of verification exhaustively examines all possible execution paths. It rigorously tries to prove or disprove the correctness of a formal model of a design. Nevertheless, due to scalability issues, formal methods are applicable only on a modular level or on the designs of a moderate size and complexity [20]. To gain confidence in the system's correctness with high complexity, such as a modern

microprocessor, industry still heavily relies on the dynamic verification techniques or simulation-based verification.

The simulation-based approach, on the other hand, is scalable but deals only with the finite set of execution paths. It can verify the system's correctness on the finite number of states based on the stimuli that get fed into the system. For the microprocessors, this stimuli is a verification code-base that usually consists of the following three: (1) simple directed unit-tests, (2) real applications, and (3) randomly generated instructions.

Chapter 3

Single core co-simulation

> Yesterday, I was clever, so I wanted to change the world. Today, I am wise, so I am changing myself.
>
> Rumi

3.1 Introduction

To be better than the state-of-the-art, we should at least start at the level of state-of-the-art. Hence, I begin by explaining the platform that we build for our experimental studies. The artifact of this work is the-state-of-the-art co-simulation framework for RISC-V cores. We called it Dromajo. By explaining Dromajo, I unfold in great details the concept of co-simulation and how to implement it.

3.2 Dromajo

Dromajo is an emulator designed for co-simulation with RTL processors that implement the RISC-V RV64GC instruction-set. Dromajo provides a simple set of APIs that allow for flexible integration for a variety of RTL processor implementations.

Dromajo enables executing applications, such as benchmarks running on Linux, under fast software simulation (17 MIPS). It can generate checkpoints after a given number of cycles, and resuming such checkpoints for HW/SW co-simulation. The results prove this to be a compelling way to capture bugs, especially in combination with randomized tests.

3.3 Checkpoint Definition

Dromajo checkpoints include the processor architectural state (registers, CSRs), the memory, but also reprogram interrupts (PLIC/CLINT), and performance counters such as the cycle and instructions executed. This is achieved by creating a small valid bootrom. Checkpoints in Dromajo consist of two parts: (a) memory image and (b) bootrom image. The created bootrom is a valid RISC-V program. It leverages the RISC-V debug spec that allows changing many supervisor registers. Since most RV64 CPUs support the RISC-V debug spec, the checkpoints created by Dromajo can be shared across different CPUs without requiring changes in the initialization beyond loading a different memory and bootrom.

Checkpoint based co-simulation allows to create portable stimulus and in-

creases productivity without the need to recompile various benchmarks. A common use is to split the Linux boot sequence in several checkpoints to speed up verification. Other advantages of using checkpoints include: (a) apply concepts of phase analysis and simulation points to capture important phases in a portable format; and (b) allows a long-running program to be checkpointed and run in parallel which reduces the simulation costs.

We could leverage the concept of *Phase Analysis* [41] and checkpoint the benchmarks at the *simulation points* [40]. We then can co-simulate, for example, SPEC benchmarks in a fast manner by loading the simulation points that represent different phases of the program.

Despite the above-mentioned benefits, one disadvantage of co-simulation with checkpoints is that the branch predictor tables, caches, TLBs, and other memory elements will start the execution from the reset state. It is problematic because we lose microarchitectural states from which the bug could potentially be manifested. We believe that Logic Fuzzer's Table Mutators can partially close this gap as we can pre-populate or randomize all the tables.

3.4 Verification Flow with Dromajo

Figure 3.1 illustrates one possible way to implement a RISC-V core verification flow using Dromajo in five steps. The whole flow can be broken down into two main parts: checkpoint generation (Steps 1, 2, 3) and co-simulation (Steps 4, 5).

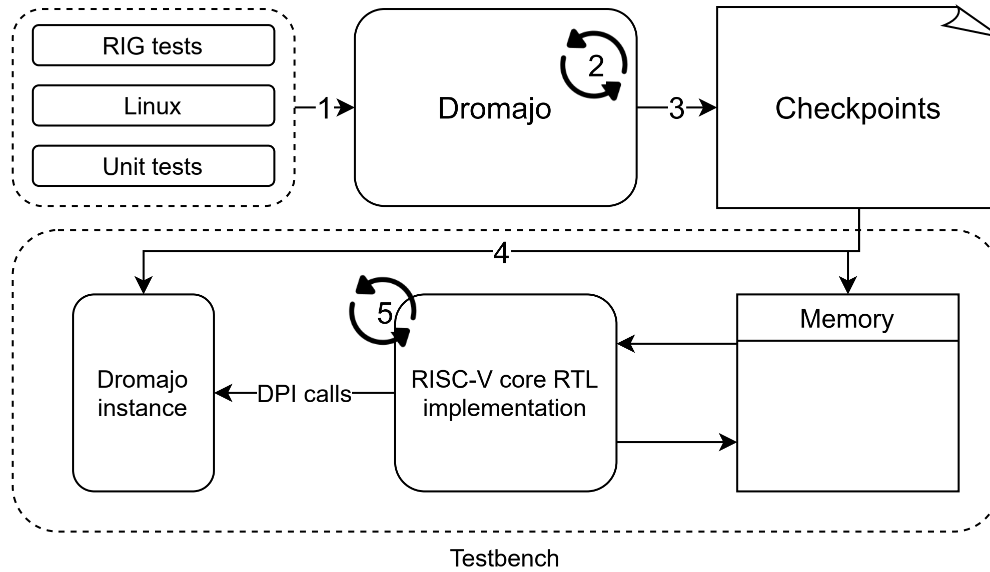

Figure 3.1: Verification flow with Dromajo

3.4.1 Checkpoint Generation

First, Dromajo accepts an arbitrary RISC-V ELF binary (Step 1). The flow has been productive when using randomly generated tests. We then run Dromajo standalone (Step 2) for a certain amount of time and dump the model's whole architectural state to a checkpoint (Step 3).

3.4.2 Step and Compare

As shown in Figure 3.1, Dromajo gets instantiated and encapsulated in the RTL as a submodule. When the simulation starts, we load the checkpoint into both models' main memories (Step 4). Dromajo instance is provided with the path to the checkpoint location. At the same time, the RTL model has to populate the main

memory and initialize the content through Verilog function like *readhex*. Once the boot code completes running, both cores will have identical architectural states.

3.5 Dromajo Integration

Dromajo is compiled into a shared library. We then link this library to a simulator and interact with Dromajo through DPI calls from the Verilog. The implementation of DPI functions is trivial. Mainly, they serve the role of wrappers for Dromajo's set of API functions. Below, we explain the only three functions that we need to call to integrate Dromajo into the verification flow.

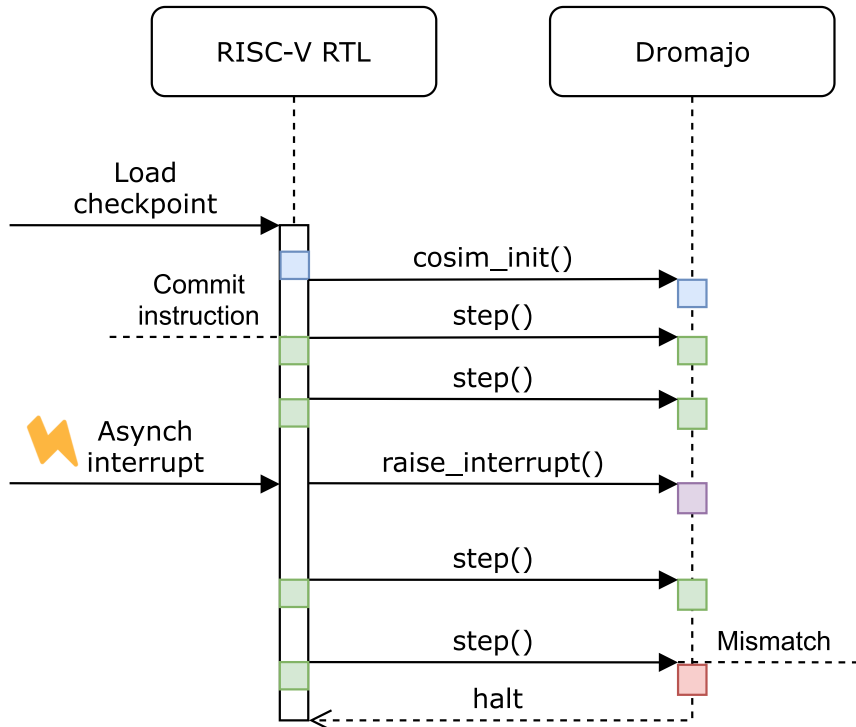

Figure 3.2: RTL-Dromajo interaction flow

Figure 3.2 depicts the interactions that happen between RTL and Dromajo during the simulation. The DPI wrapper-function *cosim_init()* is invoked within an *initial* Verilog block. In turn, the *cosim_init* invokes Dromajo's initialization API function. It passes a path to the configuration file as an argument and initializes Dromajo. The function returns a pointer to the initialized Dromajo RISC-V reference model. The configuration file contains the path to a checkpoint and the core-specific information, such as a memory map.

The DPI wrapper-function *step()* communicates program counter, instruction and store-data to Dromajo. This function should be called whenever a valid instruction is committed (Step 5). For example, when integrating Dromajo into BOOM infrastructure, the DPI can be called by implementing simple monitor logic in the Reorder Buffer module. If the instruction at the head of the buffer is valid and ready to be committed, we call the DPI. Upon invocation, Dromajo commits one instruction on its side and conducts a comparison of communicated data. The function returns a non-zero code in case of a mismatch, and we abort the execution.

Co-simulation is a synchronous process, but interrupts are not. Hence, we need a way to log that a core took an interrupt and force the control-flow inside Dromajo to do the same. It allows us to co-simulate the interrupt trap handler routines. The DPI wrapper-function *raise_interrupt()* does that. It communicates the cause and sets the trap vector in Dromajo.

3.6 Deterministic RISC-V co-simulation

One of the prerequisites of the co-simulation is a deterministic simulation infrastructure. A common RISC-V verification infrastructure tends to use Debug Transport Module (DTM) [6] to load the test binaries to RTL and generate artificial system calls.

Interestingly, our experiments show that usage of DTM brings the core into a nondeterministic architectural state which led to false-positive co-simulation mismatches. The interaction with the host device through the memory-mapped DTM is sensitive to the characteristics and utilization of the machine running the simulator. As a result, the simulation is sometimes not deterministic. Since DTM is so common, Dromajo also supports it. However, Dromajo allows creating memory and bootram checkpoints which makes usage of DTM unnecessary. Also, avoidance of DTM usage speeds up simulation as we no longer spend time uploading the binary during the simulation. We instead prepopulate the memories before the simulation start.

By using Dromajo checkpointed infrastructure, we had a fully deterministic architectural state in the evaluated cores. Independent of this work, it may be interesting work to explore a DTM 2.0 where blocking CSR operations are implemented instead of non-blocking memory accessed to bring up binaries or model fake system calls.

3.7 Related Work

Imperas Software Ltd. is a commercial company that develops virtual platforms supporting a range of ISAs, including RISC-V [33]. They claim to support step-and-compare simulation capability [32]. Imperas provide the RISC-V core *models* under the Apache 2.0. license. However the model is attached to the *simulator*, which they licensed under *OVP Fixed Platform Kit*. The difference of Dromajo with the Imperas' solution is the ability to handle checkpoints. Another difference is that the whole Dromajo is licensed under Apache 2.0.

lowRISC created a verification flow for their 32-bit RISC-V core Ibex [31]. The infrastructure that they set up executes the binary of interest both on the RTL implementation and the respective golden model. The models are completely decoupled and run the binary independently. The correctness checking happens post-completion by comparing the execution traces. In this setup, the golden model is completely oblivious to the activities happening on the RTL side. An external stimulus, such as interrupts and debug requests, will cause the traces to diverge. Therefore, this flow cannot support the instruction-by-instruction comparison in the presence of an external stimulus.

The tool proposed by Herdt et al. [18] have the co-simulation component in their flow. The novelty of their approach is that the instruction stream generator and co-simulation come in one package. They present a testbench infrastructure where instructions are generated at run-time and co-simulated endlessly. Like Ibex Core Verification flow this flow does not support the handling of the asynchronous stimuli.

Whisper [10] is the instruction set simulator developed by Western Digital. It can also be used in the co-simulation environment. The difference with Open-Cosim is that it supports only RV32, it doesn't handle interrupts, and has no support for checkpoints.

There are several general resources available for verification of a RISC-V processors. The GitHub repository developed in UC Berkeley has the set of unit tests that sweeps through the base instructions defined in ISA [13]. The RISC-V International Association has also established a Compliance Task Group [12]. Similar to [13], they only check for the basic functionality, but it is an attempt to formalize the compliance process. Currently only the RV32I ISA subset is completed. In addition, there are several open-source random instruction generators available [9, 14, 36, 38].

Finally, there is a set of works that use the term “co-simulation” in the context of heterogeneous simulation frameworks [7, 53]. In these settings, the software model is a part of a testbench and is used to drive stimuli to a design-under-test. These should not be confused with the co-simulation concept presented in this dissertation as we are describing it in the context of comparison with the golden model.

Chapter 4

Marionette Models: going beyond single-core co-simulation

> Stream input and output hate to /dev/null.
>
> — Nursultan Kabylkas

4.1 Introduction

In the context of the described state-of-the-art simulation-based microprocessor verification practice in Chapter 3, we are bringing the reader's attention to the following problem. The abstraction level of the reference model is too high to verify many critical components of the processor. The current verification infrastructures models a single-cycle machine which operates on the instruction level granularity, primarily to verify the functional correctness at the ISA level. This setup makes it impossible to

verify processor components that span their logic across several pipeline stages or even go beyond the core into caches.

For example (Case 1), the reference model completely abstracts away a branch predictor despite its importance to the performance. In the modern out-of-order cores, the branch predictor is a complex sub-system with many non-deterministic timing-dependent events whose microarchitectural state gets accessed across different pipeline stages. For instance, the branch predictor tables are read in the decode stage, they get speculatively updated at the execute stage, the updates are dropped on conflicts, etc. All these details are invisible to the verification infrastructure with the current methodology.

Similarly (Case 2), the reference model completely abstracts away the intricacies of memory request orderings involved in shared memory multi-core systems. Since modern processors implement relaxed memory consistency models, the order in which loads and stores access memory is not in the program order. The mechanism of memory request handling is a complex sub-system by itself with the hierarchy of hardware logic such as load-store queues, store buffers, and multiple levels of caches. When the request will arrive and leave the specific point in the hierarchy is a non-deterministic timing-dependent event. All these details are invisible to the verification infrastructure with the current methodology. In fact, when running shared-memory programs, the co-simulation as a technique loses its feasibility due to the data races. Simple checks at the commit stage are not sufficient. Hence, the verification of shared memory and co-simulation has always been mutually exclusive concepts.

One way to solve these issues is to build software that models the above-mentioned complexity. However, this approach nearly replicates the original RTL implementation, which makes it impractical or at the least a high-cost solution.

We propose a practical and inexpensive solution, Marionette Models (MM). Our methodology decouples complex time-dependent logic from the RTL implementation. These models are oblivious to the hardware complexity, the behavior is mimicked with simple data structures, and updates/accesses to the data structure state happen only when notified by the hardware. We draw an analogy with a marionette puppet where the complex behavior such as human body movement is reduced down to the pulling of strings. The marionette is a simple software model that mimics the human. A human is a complex non-deterministic system that acquires the knowledge of when to pull a specific string. The strings are the set of API functions that connect two systems.

To resolve the problem, a typical branch predictor's marionette is designed as a simple set of arrays and establishes the following communication points with RTL (the list may vary depending on the complexity of the branch predictor.). (1) When the branch performs (at executing stage) the RTL notifies the model to update the tables. (2) The RTL notifies the model if the update should be dropped. (3) The pipeline flushes notify the model and update some branch predictor bits like GHR. (4) When the RTL does a branch prediction, the model is queried for the outcome for a given branch, and both the RTL and model should have the same prediction.

In fact, this marionette model can be used beyond verification to explore the branch predictor's design space. The model could measure the performance impact of

experiments such as "delayed" updates or adding features such as statistical corrector. Many copies of the lightweight marionette models with different configurations could be launched simultaneously and report possible improvements or degradation.

For multi-core system verification (Case 2), the same methodology is applied. The whole sub-system's marionette is designed as a simple single array to model memory and several simple auxiliary tables to track the ordering of requests from different cores. The marionette is "hooked" to the RTL across the pipeline stages and a cache hierarchy. To name a few, (1) when the load is globally performed, typically, at the moment when the load value goes to the register file or the line is present in the L1 cache/buffers, we notify the model about the value that was seen by one of the cores. (2) Depending on the complexity of the core, the stores may merge utilizing write-combine buffers. For a simpler out-of-order core like BOOM [56], the stores are globally performed when they retire from the ROB. The detailed breakdown of the multi-cores marionette is discussed in 4.6.

By including ordering constrains like fences. The model can check that the loads get the correct value. This verifies not only the coherence but also the consistency. It even allows to verify systems that break coherence but respects consistency.

4.2 Marionette Models

Our central observation is that current co-simulation infrastructures makes it nearly impossible to incorporate many critical processor components into the simulation.

Marionette Models provide a practical solution to this issue. The methodology expands the state-of-the-art simulation based verification technique.

Marionette Models are simple notification based software models of the hardware. They are completely oblivious of the hardware complexity and updates/accesses to the micro-architectural state happen only when notified by the hardware implementation. We draw an analogy with a marionette puppet where the complex behavior such as human body movement is reduced down to the pulling of strings. The marionette is the simple software model that mimics the human. A human is the complex non-deterministic system who acquires the knowledge of when to pull the specific string. The strings are the set of API functions that connect two systems.

We present a generic methodology to develop a Marionette Model for the hardware component of interest. The methodology comprises of the following steps.

1. Marionette Skeleton Design - analysis of the hardware sub-system to extract the micro-architectural state and identify proper data-structures which holds the state.

2. Marionette Joint Design - identification of all events that alter the state of the "skeleton" as well as the events that request for the state.

3. Marionette Building - implement the model by instantiating the "skeleton" and implementing all "joints" as API.

4. String Hooking - identification of the conditions in the RTL that triggers the API calls.

5. Staging the Marionette Play - Running simulations.

To get to the final version of the model, the process might require several iterations over the listed steps to adjust assumptions, add more data structures or add more APIs.

4.3 Marionette Model of CVA6 Branch Predictor

To build an intuition, let's start simple and apply the methodology on the implementation of a branch predictor of CVA6 in-order processor [54].

4.3.1 Step 1: Marionette Skeleton Design

By analyzing the RTL implementation of the CVA6's branch predictor, it was determined that CVA6 implements a trivial bimodal two-bit-counter (2BC) branch predictor [30]. To implement the model of this predictor, all we need is the set of tables to represent the branch history table (BHT) and branch target buffer (BTB).

4.3.2 Step 2: Marionette Joint Design

This step requires capturing the main behavior of the 2BC branch predictor. The key in designing the "joints" is in identification of all events that alter the "skeleton" state, as well as the events that read the state . For this particular case, by studying the RTL implementation, we determined that the behavior of such a simple hardware component can be reduced down to four events: (1) get the predicted branch direction and target address, (2) update BHT, (3) update BTB, (4) flush the tables.

4.3.3 Step 3: Marionette Building

Once design decisions were made, a typical implementation pattern is to instantiate the "skeleton" from Step 1 as private members and all the "joints" from Step 2 are naturally translated into set of public methods (API). We implement the marionette in C++. It could be implemented in various ways and algorithms. It is up to the programmer to provide due diligence to minimize the performance and memory overheads.

4.3.4 Step 4: String Hooking

We then identify the control signals in RTL implementation of the branch predictor that correspond to read, update and flush conditions. These conditions are hooks to which we attach the appropriate API calls via SystemVerilog's DPI feature. Table 4.1 summarizes this step.

API calls	Conditions in RTL
UpdateBHT()	resolved_cf.is_vallid
	& resolved_cf.is_branch
UpdateBTB()	resolved_cf.is_valid
	& resolved_cf.is_mispredict
	& resoved_cf.is_branch
GetPrediction()	bp_valid
FlushTables()	flush_bp_i

Table 4.1: List of API functions and the corresponding conditions under which they are called. For brevity purposes, the signal names were shortened. All of these signals are defined in CVA6's frontend.sv.

4.3.5 Step 5: Staging the Marionette Play

Lastly, we run simulations with the marionette in action. The results received in this step may require loop back to previous steps to revamp the marionette. Once polished, we track deviations and uncover bugs. In fact, we could go beyond verification and use the marionette to explore branch predictor's architectural-space.

4.4 Marionette Model Integration Details

Figure 4.1 illustrates the test-bench infrastructure. Marionette Models are compiled into a shared library. We then link this library to a simulator and interact with the models using SystemVerilog Direct Programming Interface (DPI). DPI is a feature that creates interface between SystemVerilog and a foreign programming language, in our case C++. It allows the designer to easily call C++ functions from SystemVerilog.

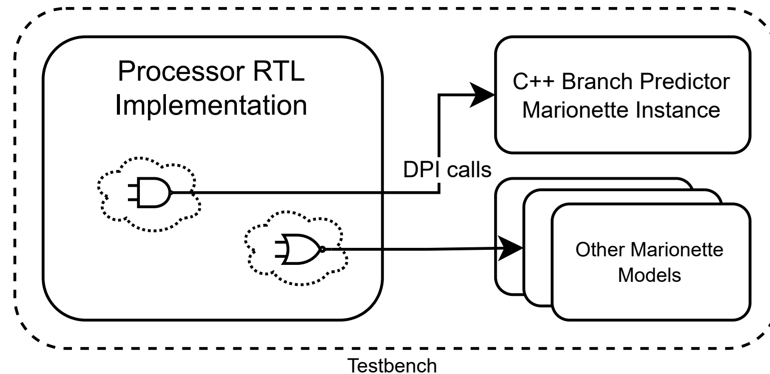

Figure 4.1: Simulation test-bench. Illustration of how the hardware components ”pulls the strings” of the Marionette Model.

When the conditions listed in Table 4.1 evaluate to true, the DPI call notifies

the model about the event. The model, in turn, makes corresponding changes to its state. Following our analogy, the hardware pulls the string to change the state of the marionette.

4.5 Why Co-simulation Does Not Work for Multi-core?

Let's take a closer look at the gaps of the state-of-the-art simulation based verification and how Marionette Models can close these gaps.

4.5.1 Inapplicability of Co-simulation on a Shared-Memory Multi-Core System

In a pipelined processor, loads retrieve values from memory at execute stage, but the values get written to the registers at commit stage. Figure 4.2a visually illustrates this scenario. We want to emphasize that there is a Δt between bringing the data into the pipeline and the actual commit of that value to the architectural state of the processor. The state-of-the-art co-simulation techniques [24, 31, 32] work if and only if we can guarantee that no other event in the system alters the state of the memory within the Δt. The reference model requires this guarantee as it treats memory operations as a single step events. In other words, the reference model assumes that Δt is zero.

This guarantee is implicitly provided in the uni-processor settings as there is only one source of loads and stores in the system. Hence, we can apply the co-simulations. Nevertheless, as shown in Figure 4.2b, the guarantee can no longer be provided when we include other sources of reads and writes. The co-simulation, as

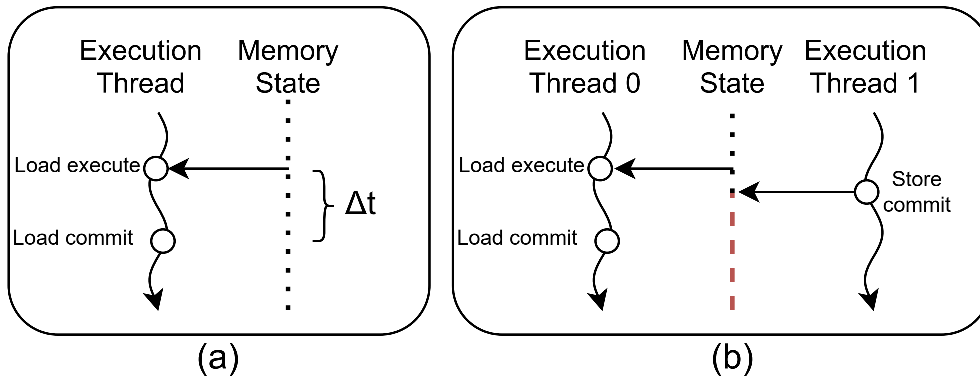

Figure 4.2: Execution threads of memory operations.

it is applied today, becomes infeasible because the reference model is unaware of the intermediary states of memory operations. To understand the issue more clearly, let's consider the simplistic test code listed in Table 4.2 running on a dual-core system. The code illustrates a common shared memory program pattern when one core is loading data from the given memory address, while the other core is writing.

Core 0	Core 1
(i1) $r1 \leftarrow$ [x]	(i2) [x] $\leftarrow 1$

Table 4.2: Simple test code

Figure 4.3 illustrates how these two instructions are executed on the system. The instructions go through a typical out-of-order processor pipeline. The definitions for the acronyms presented in the figure are summarized in Table 4.3. Without loss of generality, assume that some of the pipeline stages lump in the functionality of common stages, such as register renaming, dispatching, instruction issue, etc.

Let's draw our attention to a series of events happening in the pipeline high-

Pipeline stage name	Acronym
Fetch	F
Decode	D
Wake-up/Select	WS
Register file access	RF
Execute	X
Write Back	WB
Commit	C
Stall the pipeline	–

Table 4.3: Pipeline stages of a simplified out-of-order processor.

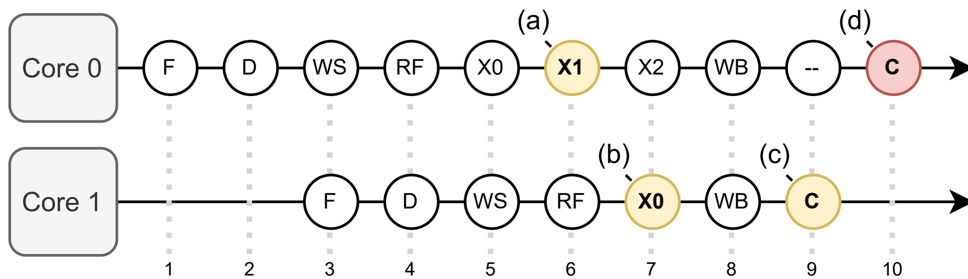

Figure 4.3: Timing diagram of a multi-core system executing memory operations (x-axis is time). (a) to (d) emphasize the events occurring at the highlighted stages.

lighted in Figure 4.3.

- Event (a): core 0 accesses data cache, it hits and reads data from memory location x.
- Event (b): core 1 locally performs the store, i.e. the data is locally placed into the core's store queue.
- Event (c): core 1 commits the store, i.e. the store writes the memory.
- Event (d): core 0 commits the load, i.e. this is the point of no return. The value

that was written back in the WB stage is irreversible.

At event (d), the RTL will signal the reference model to commit the load on its end and compare the values. The problem is that Core 0 requests the comparison of the data that was loaded at event (a). Between the events (a) and (d), event (c) overwrote the memory. In other words, by the time load's data got to the commit stage, the store of the other core had enough time to overwrite the value. Hence the co-simulation fails despite the fact that the system is behaving as it supposed to. Note that the co-simulation is not capable of handling the simplest shared-memory program.

To solve this issue, we must somehow expose to the co-simulation infrastructure an exact moment in the pipeline when the loads and stores perform.

4.5.2 Incapability of the Co-simulation to check for Memory Ordering Specification

In the uni-processor world, ordering of memory operations is quite intuitive – a load must get the value that was written by the last store. This thinking model is straightforward and does not leave any room for ambiguity. The processor implementation may even heavily optimize the memory operations for performance, i.e. place the stores to the store buffer, execute them out of order or speculatively. As long as the loads see the last stored value when they commit, the intricacies of memory instructions can be safely abstracted to a single step operations.

In the multi-processor world, however, the borders of the above mentioned unambiguous simple model get blurry. When two or more processors reading and writing

a shared memory, due to a hierarchy of various buffering, the value written to a memory location by ”the last store” may not be consistent across different processors. In other words, what the loads see on different processors may diverge. For example, Figure 4.4 illustrates one possible execution of the litmus test listed in Table 4.4. This particular execution sets $r1 = 1$ and $r2 = 0$. We want to draw reader's attention to the following two observations. First, the values that the loads see depends on timing dependent events and the execution flow of a particular run. In other words, there many different possible values that the loads can see. Second, out of several possible values that loads can see, we should specify which ones are legal. In other words, is ordering of memory operations complies with the Memory Consistency Model of the ISA. For instance, what is illustrated in Figure 4.4 is a violation of the SC [29] and TSO [43] memory consistency models because the stores became visible to the global memory out of program order.

Thread 0	Thread 1
(i1) [x] $\leftarrow$ 1	(i3) $r1 \leftarrow$ [y]
(i2) [y] $\leftarrow$ 1	(i4) $r2 \leftarrow$ [x]

Table 4.4: Litmus test

To reason about the memory model correctly, the co-simulation infrastructure should know what does ”the last store” mean and be familiar with the rules of the given memory consistency model. To enhance the co-simulation with the memory operation ordering checking, we must somehow expose to the co-simulation infrastructure intermediate states of the memory operations and check for constraints imposed by memory consistency model on the fly.

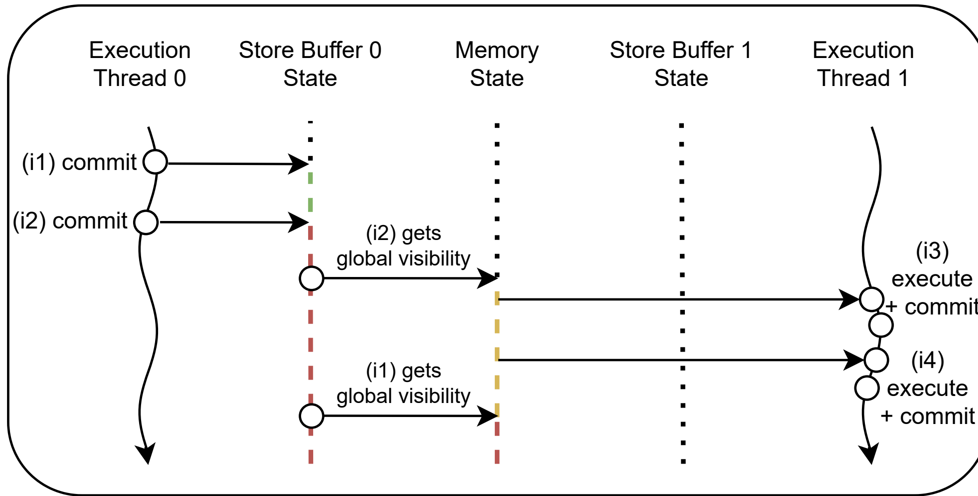

Figure 4.4: Execution threads of memory operations with intermediate buffering.

In the following section, we will demonstrate exactly how the Marionette Model for the shared memory can be built to close the above mentioned gaps.

4.6 Building Shared Memory System's Marionette

To build the Marionette Model for the shared memory system, we apply the steps of the methodology described in Section 4.2.

4.6.1 Step 1: Marionette Skeleton Design

In this step, our task is to analyze the hardware sub-system to extract the micro-architectural state and identify proper data structures to model the hardware component. By looking at the shared memory system from the bird's-eye-view, we observed the following characteristics:

1. The system maps addresses to data when accessing memory.
2. The system has a hierarchy of value buffering (store queue, store buffers, MSHRs, etc.)
3. The system imposes rules on the ordering.

To build a simple and abstract model that exerts behavior with the listed characteristics, we chose the following data structures:

1. A map that maps memory addresses to data. Essentially, this is the model of the main memory.
2. A set of queues to buffer the intermediate states of memory operations and also to keep track of the order. The queue is instantiated per core just as it happens in the hardware.

Note that this is only one possible ways to implement the model. Given the characteristics, it is up to the diligent engineer to transcribe this characteristics into a specific implementation.

Figure 4.5, visually illustrate the skeleton of the shared memory system's marionette. A quite simple model: several queues instantiated based on the number of cores and a single memory that maps addresses to the corresponding values.

4.6.2 Step 2: Marionette Joint Design

The key in designing the "joints" is in breaking down the behaviour by identification of all events that alter or read the state of the "skeleton". The high-level

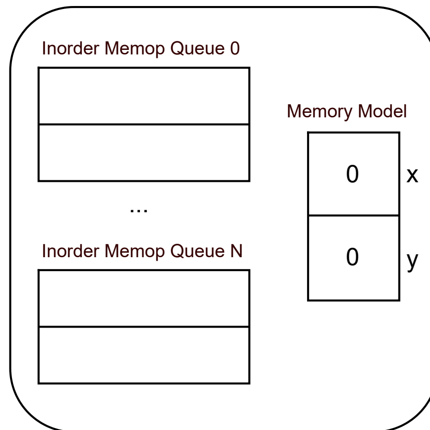

Figure 4.5: The “skeleton” of the shared memory system’s marionette.

behavior could be captured by the events which are illustrated in Figure 4.4. The horizontal arrows in the Figure are the moments in execution when the state is read or written:

- **A store writes the intermediate buffer**. In other words, following the literature nomenclature [34], the store is locally performed. This means that the effect of the call is only local, i.e. the loads from the other cores do NOT see this effect. In Figure 4.4, this event is illustrated by the arrow that goes from Execution Thread 0 to Store Buffer 0 State.

- **The local effect of the store gets propagated to the global memory**. In other words, following the literature nomenclature [34], the store is globally performed. This means that the effect of the call is global, i.e. the loads from the other cores DO see this effect. In Figure 4.4, this event is illustrated by the arrow that goes from the Store Buffer 0 State to Memory State.

- **A load of a given processor reads the value**. To be consistent with the nomenclature, we refer to this event as "performing the load". In Figure 4.4, this communication point is illustrated by the arrow that goes from the Memory State to the Execution Thread 1. Note that the arrow could go from the Store Buffer 1 State to the Execution Thread 1 if there were stores that locally performed previously to the same address. Note, this event takes place at the execute stage of the load (not commit). As was discussed in Section 4.5.1 and as illustrated in Figure 4.2a, we need to capture this event to guarantee Δt of zero, which is requirement for the correct co-simulation.

The above-listed events cover the high-level behaviour. Depending on the complexity and the performance optimization techniques used in the processor, we may need to capture more events. My experimental studies targeted BOOM's shared memory system [56]. Hence, I had to capture the following events:

- **A memory operation gets removed from the queue**. We refer to this event as "nuking". BOOM is a pipelined processor with a branch predictor. Hence, the memory operations that got into the pipeline due to misprediction must be nuked at the branch resolution stage.

- **A store gets merged with the previous store to the same address**. We refer to this event as "merging the store". This event needs to be captured if the processor implements buffering in the intermediate structures like MSHR [27] or purposeful store buffering in write-combine buffers [21,39] past the commit point.

For instance, if we were to commit several stores to the same cache line before the cache line is present in the cache, we must merge all those stores locally. Then, when the cache line is present in the data cache, all those stores are made visible globally all at once.

Finally, there is also a need in addition of events that play auxiliary role, such as addition and the removal of the memory operations into and from the data structures.

4.6.3 Step 3: Marionette Building

Once design decisions were made, a typical implementation pattern is to instantiate the “skeleton” from Step 1 as private members and all the “joints” from Step 2 are naturally translated into set of public methods (API). We implement the marionette in C++. It could be implemented in various ways and algorithms. It is up to the programmer to provide due diligence to minimize the performance and memory overheads. Our implementation of the Marionette Model has been open-sourced and is available on GitHub [48].

4.6.4 Step 4: String Hooking

Once the model is ready, we analyze the control signals in RTL implementation of the BOOM to construct logical conditions that would signal occurrence of the events in Step 2. These conditions are “hooks” to which we attach the corresponding API calls via SystemVerilog’s DPI feature. Table 4.5 summarizes this step.

API calls	Conditions in RTL
InsertMemop()	rob_io_enq_valid & rob_io_ready & (rob_io_enq_uops_uses_ldq | rob_io_enq_uops_uses_stq)
StoreLocalPerform()	stq_X_bits_addr_valid & stq_X_bits_data_valid
StoreGlobalPerform()	dataWriteArb_io_out_valid | metaWriteArb_io_out_valid
LoadPerform()	io_wb_resps_valid & write_back_uses_ldq
NukeMemop()	negedge (stq_X_valid) | negedge (ldq_X_valid)
StoreMerge()	stq_X_bits_succeeded

Table 4.5: List of API functions and the corresponding conditions under which they are called. For brevity purposes, the signal names were shortened.

4.6.5 Step 5: Staging the Marionette Play

Lastly, we run simulations with the marionette in action. The results received in this step may require looping back to previous steps to adjust the model, i.e. change/add data structures or extend the API to support more events. Once polished, we track deviations and uncover bugs.

Figure 4.6 illustrates a timing diagram of a simplified out-of-order pipeline along with the API invocations. The tables on the left side of the Figure are classical pipeline timing diagram of two cores where a row corresponds to a particular instruction and a column correspond to a particular clock cycles. Consequently, each cell in the table represent the instruction's location in the pipeline at a specific time. For instance, "WS" in the first row of core 1, indicates that "add" instruction is in core 1's wake up/select stage at clock cycle 5. For the full list of abbreviations refer to Table 4.3.

The arrows in the Figure 4.6 that go from a specific stage in the pipeline to the

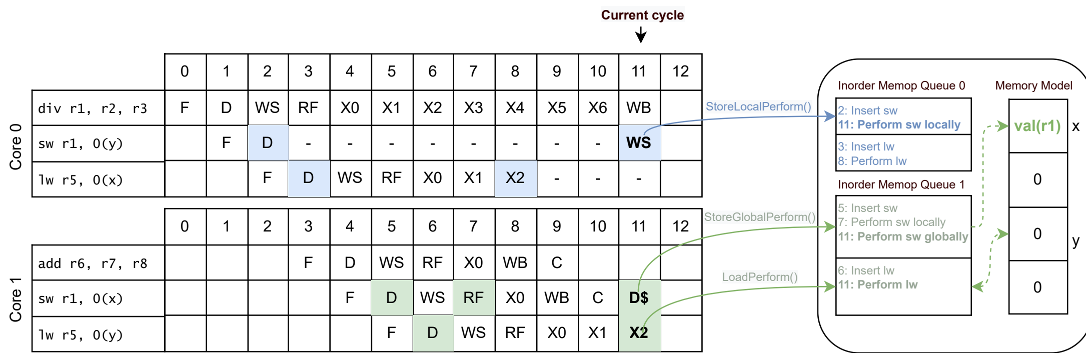

Figure 4.6: Pipeline timing diagram of a multi-core system displaying activities happening at clock cycle 11

Marionette Model on the right of the Figure represent API invocations at clock cycle 11. Following our analogy with the marionette puppet, this illustrates how RTL “pulls the strings“ of the marionette at clock cycle 11. The breakdown of the activities at the clock cycle are listed below.

- sw instruction of core 0 invokes StoreLocalPerform() at wake-up select stage. The store gets woken up because the dependant data from the previous div instruction becomes available. Once, both data and the address of the store instruction are ready, this store becomes visible locally. In other words, local loads to the same address will forward the value of the store.

- sw instruction of core 1 invokes StoreGlobalPerform() from the data cache right after it was committed in clock cycle 10. This means that it was cache-hit and the cache-line was in modified state.

- lw instruction of core 1 invokes LoadPerform() from the execute-2 stage. The

second stage of execute is typically when the value is read from the cache and gets bound to a physical register. At this point, we perform the comparison of the data that was read in the RTL with the value in the model.

Chapter 5

Logic Fuzzer: enhancing the state-of-the-art

> And tire yourself out, because it makes life worth living! I have seen that water stagnates when it stands still, yet when it runs it is sweet and pure.
>
> ---
> Imam Al-Shafi'i

5.1 Introduction

Current state-of-the-art verification infrastructure does not guarantee that the processor is bug-free when sending the design for fabrication. The study on verification trends shows that fewer companies achieve first silicon success due to increased

complexity and need costly re-spins before production. Despite the immense amount of co-simulation and achieved high coverage, 40% of the reasons that cause a re-spin are functional bugs that escape to silicon [17]. These bugs get exposed only during the silicon validation or even worse at the end-customer. The verification engineers call these bugs *outlier bugs* or *simulation resistant super bugs*. The outlier bugs exposed neither by random instruction streams nor by directed tests because the sequence of events for the bug to occur is too complicated. They can only "be exposed by exercising the design outside its normal flow or operating parameters" [5].

In this Chapter, I explain Logic Fuzzer, a noval technique that brings the processor's execution outside of its normal flow and increases the chances of finding outlier bugs in the simulation phase. The key in simulation-based verification is to develop the code sequence that will bring the processor into a buggy microarchitectural state that results in an inconsistent architectural state with the golden model. The Logic Fuzzer increases the microarchitectural states reached *without* the engineering effort of developing new code sequences, hence increasing the productivity of time spent on the test creation. The overall idea is to randomize states or control signals in the DUT that does not affect the correctness. For example, the re-order buffer (ROB) may assert a full or stall signal even when it is not full. It is also possible to change the branch predictor tables at any given moment or even insert the instructions in the mispredicted path. The Logic Fuzzer can change the number of cycles but does not corrupt the functionality or the program order. Our results show that atypical microarchitectural states created by Logic Fuzzer expose more bugs during the simulation phase.

Next, I discuss several fuzzing techniques and their implementation details.

5.2 Logic Fuzzer

5.2.1 Congestors: A Case for Fuzzing

The simplest type of LF is a congestor. As shown in Figure 5.1, a simple example of a congestor is an or-gate inserted at the full signal of a FIFO module. The full signal gets activated even though the condition for it to become full has not been satisfied. The congestor is then randomly activated, which results in artificial back-pressure. We could also locate the congestor at busy signals and ready-valid handshake signals.

We claim that the inserted logic stirs up the execution while running the code. We can prove this by showing that after inserting the congestor logic and running the same tests, the design is experiencing different behavior, and we can observe the new activity. We can observe this by measuring the *Toggle Coverage*. The signal is said to be *toggled* if its value switched $0 \rightarrow 1$ and $1 \rightarrow 0$ at least once while executing the test. Toggle coverage is one of the proxy metrics that is used both in industry [45] and academia [28] to gain confidence about the correctness of the design-under-test.

For example, in the case of BOOM, we inserted a congestor at the *ready* signal of the Reorder Buffer. We then randomly pulled the *ready* signal low at the moments when the ROB was, in fact, ready. As a result, 12 additional signals toggled in the *frontend* module, 40 signals toggled in the *core* module, and 32 signals toggled in the

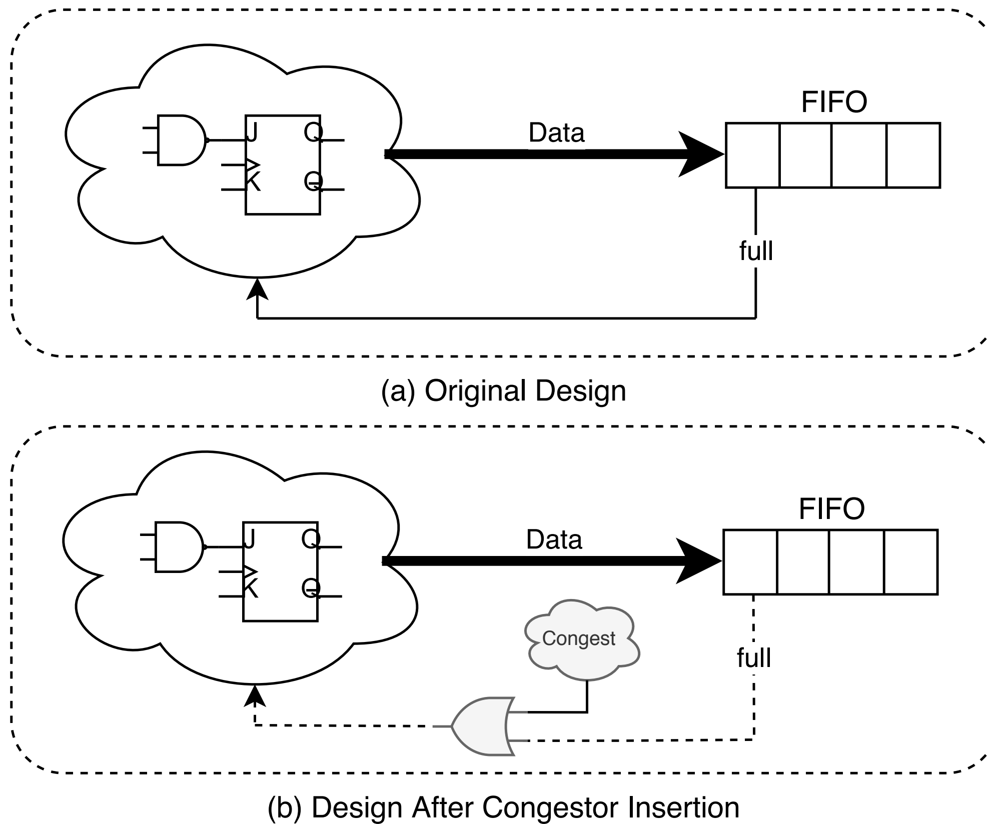

Figure 5.1: Congestor Logic Fuzzer placed at the FIFO's full signal

load-store-unit.

For instance, according to the comments in the RTL, the signal that got activated in the load-store-unit (*execute_ignore*) "ignores the next response that comes from memory and replays it." Another example is *edge_inst* signal in the *fetchcontroller*, which gets activated when "first instruction in the bundle is PC-2."

We demonstrated that single congestor can activate logic pieces that had not been touched when running over 200 tests. Note that for this simplistic example, the usage of toggle coverage was sufficient to capture and prove that Logic Fuzzer creates

additional activity. We discuss some drawbacks of the coverage metrics in 6.3.2.

5.2.2 Table Mutatators

Table mutators allow RTL memories to be mutated. For example, the tables of branch predictors can be freely fuzzed at any given moment as they must not affect the correctness of the running program. Other examples are the random invalidation of the cache or TLB entries or the value fuzzing of already invalid entries.

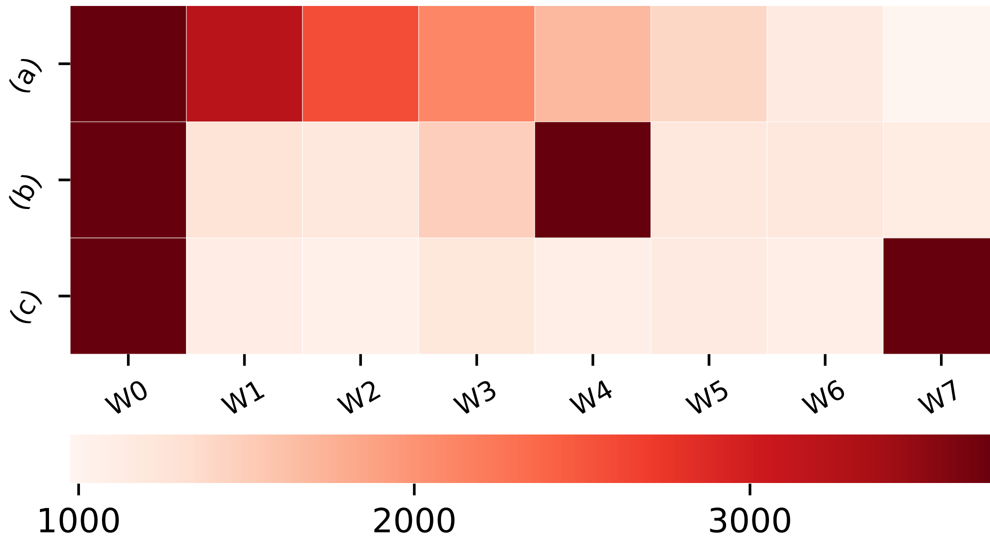

Figure 5.2: CVA6's L1 cache way/bank utilization without (a) tag array mutation and (b)(c) with tag array mutation

Taking an example from the application to CVA6, Figure 5.2 illustrates L1 cache way/bank utilization (stores only) when running over 50 random tests that were generated with Google's *riscv-dv* tool [14]. We run this set of tests three times. The first run, row (a), shows the regular run with the table mutation off. As can be seen, given the memory locations that program requests, CVA6's way selection logic gives

preference to way 0. In the verification phase, an engineer may notice this fact and might decide to stress underutilized ways. Traditionally, the engineer would have to regenerate the binaries by configuring the random instruction generator to provide the memory requests in a specific manner. The problem with this approach is that this may be a time-consuming process and will require delving into the cache replacement policies' details. Besides, the tool may not even support constraining the address generation.

The second and the third run, rows (b) and (c), illustrate that we can mutate the tag arrays and the valid bits to stir all the cache accesses to the bank of interest with the minimum amount of effort. To be specific, we edited five lines of code on the RTL side to replace tag arrays with the wrappers that access Table Mutators through DPI and the simplistic twelve-line method implementation in Table Mutator class that mutates the entries to stress the cache bank of interest.

5.2.3 Stressing mispredicted path

The branch predictor is a significant part of the design that has a significant effect on modern processors' performance. The advances in the branch prediction research have reached levels of accuracy exceeding 95%. However, from the verification standpoint, this means that the mispredicted path may be overlooked.

It is important to test all of the instructions in the mispredicted path as some may have side effects. Figure 5.3 shows the instruction coverage in the mispredicted path. The y-axis of the plot represents the number of unique RISC-V instructions that were speculatively allowed into the pipeline and eventually flushed due to the correct

branch resolution. After running over 200 tests on CVA6, we see that the coverage does not reach even a 60% level. The reason is that the instruction sequence in the program exhibit nonrandom behavior and the same instructions get into the mispredicted path repeatedly. With the fuzzing, we can insert any instruction into the mispredicted path regardless of the binary. Not only can we test 100% of the instructions, but we also can reach that coverage level earlier.

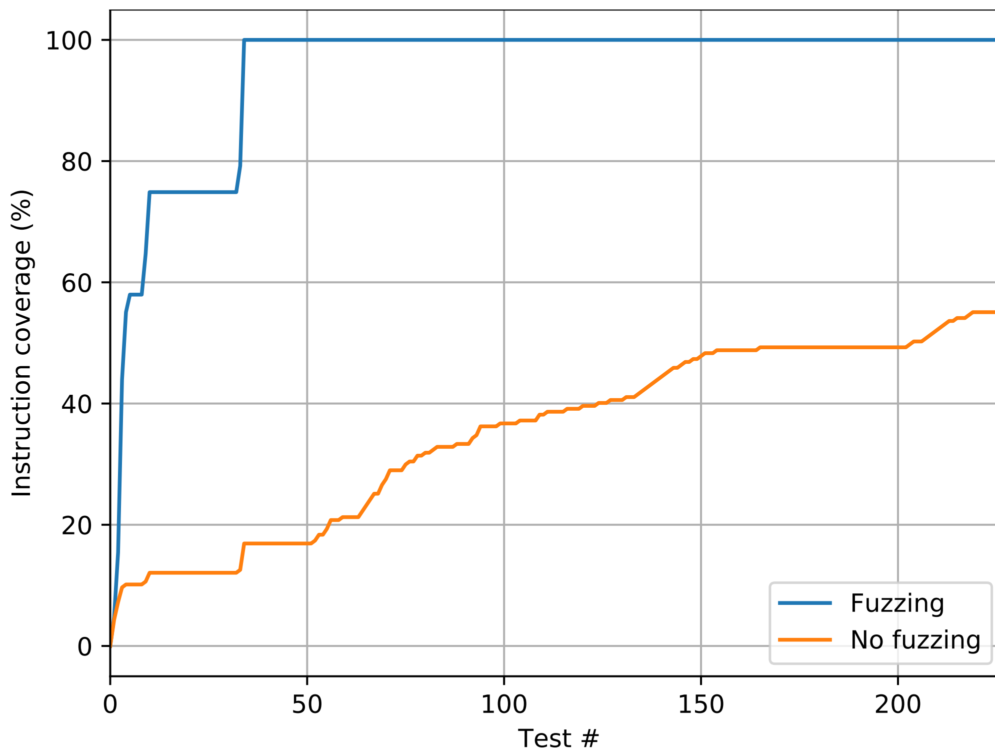

Figure 5.3: The coverage of instructions that were in CVA6's mispredicted path

Another interesting question to answer is: what will happen if speculative execution mechanisms will start generating instruction addresses that are atypical. Figure 5.4 is a scatter plot where each data point represents a Branch Target Buffer's

(BTB) prediction of the PC address in a given test. The red marks are the PC predictions from when no fuzzing was enabled. We can see that the predicted addresses are within a narrow range. On the one hand, by design, the BTB is supposed to provide predictions based on the history of resolved branch target addresses. Hence, it is constrained to the address range that is encoded in the *.text* section of the *elf* file. On the other hand, the processor must be robust enough to handle non-typical cases. The data points illustrated in blue circles are the BTB's predictions when running the same tests with the fuzzing enabled. It is possible to fuzz BTB entries and provide falsely predicted addresses to a broader range or even provide random addresses at runtime. These scenarios can potentially create an iTLB page faults on the mispredicted path. The same technique can be applied to Return Address Stack.

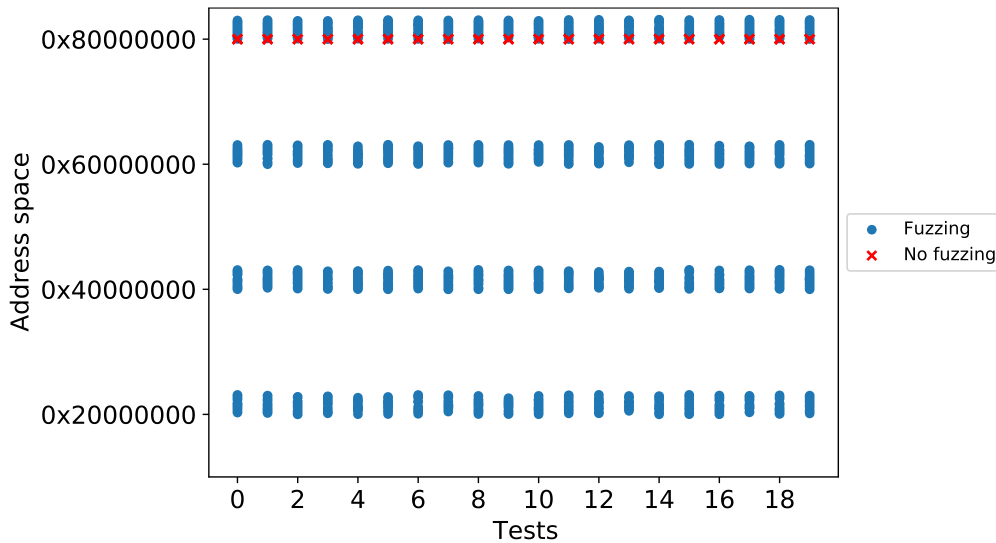

Figure 5.4: Instruction address ranges retrieved from Branch Target Buffer while executing random instructions.

5.3 Can the LF's states ever happen in the real world?

Logic Fuzzer could create a microarchitectural state that no program could ever reach. Nevertheless, the co-simulation failures exposed by fuzzing are potential bugs. They serve as a red flag for the engineer and must be proved or disproved to be a bug. The bugs presented in this dissertation were all confirmed by the designers.

5.4 Logic Fuzzer Implementation

The Logic Fuzzer as a concept can be implemented directly in the RTL code. However, to make the Logic Fuzzer integration clean, systematic, and configurable, we embedded the LF into the existing Dromajo infrastructure. We extended the base set of APIs to provide access to the fuzzers from the RTL through the DPI calls. The fuzzers are configured by Dromajo's JSON configuration file.

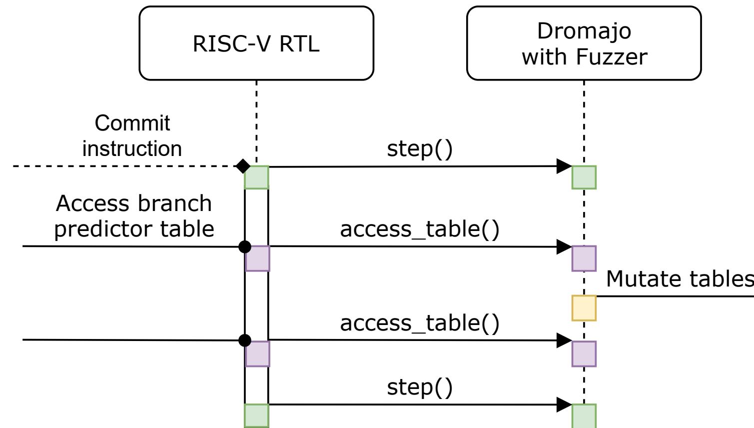

Figure 5.5: RTL-Fuzzer interaction flow

Figure 5.5 illustrates the interaction between the RTL and the fuzzers. The

diagram demonstrates how the RTL implementation is accessing the table mutator of the processor's branch predictor. The fuzzer object in the Dromajo is configured to allocate the table that has the same size as the branch predictor. On the implementation side, instead of accessing the RTL memory model, it accesses the table from the fuzzer through the DPI. As the simulation is running, the tables are fuzzed randomly or with specific patterns.

For the congestors, the verification engineer first needs to identify all of the design's congestible points. The identification of these points can be consulted with the designer. After we have the list of congestible signals, the fuzzer object is configured to create the same number of congestor objects. Each congestor's period and random seeds are configured in the JSON file.

To address the productivity issue, we implemented a proof-of-concept automatic insertion of congestors into BOOM core. It allowed us to insert congestors by merely annotating the signals in RTL (one line of code per congestor). We achieved this by using *Chiffre* by Eldridge et al [11] that automatically instruments hardware systems written in Chisel [4]. Their work leverages FIRRTL compiler that allows traversal and transformation over the intermediate representation(IR) of the digital circuit with *passes* [22]. It automatically breaks the annotated signal and inserts the congestor in between. Because *Chiffre* can only work with hardware descriptions written in Chisel, this experiment was limited to BOOM core. We are currently applying the described concept using tools capable of transforming IRs generated from Verilog [50].

To insert random instructions into the mispredicted path, we make use of the

table mutators. We replace the instruction cache tag and data arrays with the table mutators. The tables are written and read by the instruction cache logic through the DPI in the same manner as shown in Figure 5.5. We then force the Branch History Table to provide a *taken* prediction, and we force the Branch Target Buffer to provide the address with a specific tag. The fuzzer tables are then programmed to provide a random instruction stream when it sees that specific tag.

5.5 Related Works

5.5.1 Input-stimuli fuzzing

Inspired by software verification techniques, several works adapted the methods for hardware verification. RFUZZ transferred the concept of American Fuzzy Lop to hardware [28]. Trippel *et al.* [47], on the contrary, explore the methods to transfer the design to the software model and apply well-established software fuzzing techniques in software domain. The technique in the PyMTL infrastructure adapted Hypothesis Testing [42]. It is property-based testing that requires to construct assertions that must always hold. The technique then tries to find the minimal possible example that breaks the assertion. Similar method presented in [35]. These techniques should not be associated with Logic Fuzzer as all of them stress the DUT externally having an "outside-in" approach. The LF proposes an "inside-out" approach, meaning the actual RTL logic is fuzzed wherever possible.

5.5.2 Fault Injection

On a surface level a parallel can be drawn between the concepts of Logic Fuzzer and Fault-Injections [11]. A fault is a physical defect or a flaw that may occur in a hardware system. As the name suggests, Fault-Injection is a simulation-based procedure when the fault is injected into the system on purpose. The simulation is then run with the fault and the behavior is observed. The idea is to prevent the system from complete failure, even in the presence of faults. The overlapping concept with the LF is the purposeful injection of logic into the system that changes the behavior.

Nevertheless, the fundamental difference is that LF makes sure that the inserted logic does not have any side effects on functionality. On the one hand, system failures detected by fault-injection are analyzed. As a result, preventive or corrective actions are proposed. On the other hand, system failures detected by Logic Fuzzer are flagged as potential functional bugs.

Chapter 6

Evaluation

Knowledge without action is
wastefulness and action without
knowledge is foolishness.

Imam Ghazali

6.1 Evaluation Methodology

6.1.1 RISC-V Cores

The proposed verification tools, *i.e.*, Dromajo co-simulation enhanced with Logic Fuzzer, were evaluated on three open-source RISC-V processors. The details about each tested core are listed below and summarized in the Table 6.1.

6.1.1.1 CVA6

Previously known as Ariane and developed in ETH Zurich. The development and maintenance has been transferred to OpenHW Group. It is written in SystemVerilog. CVA6 is a 6-stage, single issue, in-order core which implements the 64-bit RISC-V instruction set. It is capable of booting Linux and was taped out in 22nm technology [54].

6.1.1.2 BlackParrot

Joint work of the University of Washington and Boston University. It is written in SystemVerilog. The BlackParrot is a single issue, in-order core which implements the 64-bit RISC-V instruction set. BlackParrot was written in SystemVerilog. It is capable of booting linux and 4-core configuration of BlackParrot was taped out in 12nm technology [3].

6.1.1.3 BOOM

Developed and maintained at UC Berkeley's *Berkeley Architecture Research* group. It is written in Chisel hardware construction language. It is a generator that can be configured to generate verilog BOOM designs with various levels of complexity. The generated cores implement the 64-bit RISC-V instruction set. We used default *MediumBoomConfig* which is a 2-wide, out-of-order core [56]. One of the more complex configurations of the BOOM was taped out in 28nm technology.

Features	CVA6	BlackParrot	BOOM
Execution	in-order	in-order	out-of-order
Issue width	1	1	2 (MedConfig)
Extenstions	RV64GC	RV64G	RV64GC
Priv. modes	M, S, U	M, S, U	M, S, U
Virt. memory	SV39	SV39	SV39

Table 6.1: Summary of the cores used for evaluation

6.1.2 Evaluation Metrics

I evaluate the proposed tools and methodology based on the number of bugs found or by the feature that tools enabled. The evaluation of Dromajo and Logic Fuzzer was done on single core configuration. We first run a set of binaries on the base setup, i.e. testbench infrastructure with only Dromajo enabled. We then run the same set of binaries with enabled Logic Fuzzers, which expose additional bugs. The evaluation of the Marionette Model is done by explaining the features it enabled.

6.1.3 Test Binaries

We used verification binaries from two available resources: RISC-V ISA tests [13] and random instruction streams generated with Google's riscv-dv tool [14]. Table 6.2 summarizes the simulated test binaries that we run to get the presented results.

Core	No. of ISA tests	No. of random tests
CVA6	228	120
BlackParrot	215	150
BOOM	228	120

Table 6.2: Summary of the simulated tests

For the evaluation of the Marionette Model we ran Linux on CVA6 and custom

prepared program on BOOM that heavily exercises the shared memory system.

6.2 Evaluation

6.2.1 Main Results

We evaluate the effectiveness of the tools on three RISC-V cores: CVA6, BlackParrot, and BOOM. We summarize our bug findings in Table 6.3 and Table 6.4. Dromajo by itself found a total of nine bugs. The enhancement of Dromajo with the Logic Fuzzer increases the exposed bug count to thirteen. Note that we did not create additional tests when enabling Logic Fuzzer. It used the same set of tests listed in Table 6.2 to expose additional bugs.

I demonstrated that the presented tools are capable of exposing hardware malfunctions that could prevent any complex software from running correctly. I also provide interesting observations related to Operating Systems (OS). Three RISC-V cores that we used for evaluation claim to boot and run Linux. More than half of the bugs found were OS related. Interestingly, a "well behaved" Linux will not have excercised most of the bugs. Our results show that being able to boot and run Linux is far from saying that the core is verified.

6.2.2 Dromajo Evaluation

Table 6.3 summarizes the bugs exposed by merely integrating Dromajo into the design of open-source cores. Below is the detailed explanation of each bug.

Bug ID	Core	Short description
B1	CVA6	incorrect update of *prv* bits in *dcsr* register
B2	CVA6	incorrect integer division
B3	CVA6	stval CSR is written on ecall
B4	CVA6	mtval CSR is written on ecall
B5	BlackParrot	integer divide, incorrect handling of sign-extension
B6	BlackParrot	no exception handling on some illegal instructions
B7	BlackParrot	least-significant-bit not cleared on *jalr* instruction
B8	BlackParrot	speculative long latency instructions commit
B9	BOOM	incorrect mtval CSR value on traps

Table 6.3: Summary of the bugs exposed by Dromajo

Bug ID B1. This bug is the result of incorrect update logic implementation of debug control status register and was exposed by Dromajo. The execution divergence occurs right after *dret* which should jump to the PC indicated by *dpc* CSR and in the privileged mode that is indicated by *prv* bits in *dcsr* CSR. Dromajo starts execution in the user-mode while CVA6 starts executes the following instruction in machine-mode. According to the designer, the confusion came from the fact that the core should update *prv* bits to the current running privileged level when entering debug mode.

Bug ID B2. This bug is in CVA6's integer divide unit. The unit fails to properly handle some corner case divide (*div*) and remainder (*rem*) instructions; Dromajo caught the mismatch when cores were executing division of -1/1. Dromajo committed a correct result by assigning -1 to the destination register, while CVA6 committed 0.

Bug ID B3. According to RISC-V ISA, *stval* CSR is written exception-specific information when the processor traps into a supervisor-mode. The ISA explicitly specifies when and which information must be written to the register. Dromajo catches a mismatch inside the exception handler when reading the value of *stval* due to incorrect

setting.

Bug ID B4. This bug brings about similar implementation inconsistancy within the ISA specification that is described in Bug ID B3 (6.2.2). The difference here is that *mtval* control status register is written incorrect value.

Bug ID B5. This bug is in the BlackParrot's integer divide unit. The co-simulation failed when the BlackParrot commited *divw* insruction, which is the 32-bit integer division. The bug manifests on the *remw* instruction as well. The *divw* and *remw* are signed instruction. The proper implementation should divide the lower 32-bits source registers by treating them as signed numbers, but BlackParrot's integer divide unit implementation was treating the operands as if they were unsigned.

Bug ID B6. This bug is in the BlackParrot's instruction decoder. The decoder did not trap an invalid instruction, but passed it down the pipeline. The machine code that BlackParrot decided not to mark as an illegal was the the binary word similar to the encoding of *jalr* instruction. The only difference was that sub-opcode *func3* was not equal to zero. The ISA defines *jalr* instruction with the subopcode of zero. The decoder had not perform any checks on *func3* bits at the time of the bug detection. Although the mismatch was detected for the case of *jalr*, in general, all instructions with invalid subopcodes should trigger an illegal instruction exception.

Bug ID B7. The bug is related to the calculation of the target address of the control flow instructions. The RISC-V ISA explicitly requires that the least significant bit of the calculated address by *jalr* instruction is cleared. The clearing of the bit was not implemented in the BlackParrot. Hence, Dromajo flagged the PC mismatch after

the execution of *jalr* instructions. According to the designer, the confusion came from the fact the *jalr* instruction is encoded differently than *jal* and branch instructions.

Bug ID B8. This bug is the result of an incorrect poison bit setting. Dromajo flagged a mismatch on the load instruction. Dromajo trace analysis proved that there was only a single store-instruction to writing to the memory address. However, the following load-instruction brought a different value from what had been written. According to the designer, the values were overwritten by long latency instructions that were marked for flushing. The bug would manifest when the pipeline flushed on exceptions. Then at some point, the long latency instruction completed and allowed write-back due to the invalid poison bit.

Bug ID B9. This bug gets exposed when running a random instruction stream and Dromajo flags a mismatch when reading *mtval* CSR. The cores start running the binary in an M-privileged mode to execute the setup instructions, such as a series of CSR writes. Including the write to the CSR *mepc*, which gets set to *0x196*. The last instruction of the setup code is *mret*, whose execution supposed to change the privileged mode and revert the program flow to the instruction memory address pointed by *mepc*. Nevertheless, *mret* throws an exception because of the instruction page fault. According to ISA, this fault must set the *mtval* value to the address of the instruction that caused the exception. However, when we read the value of this CSR in the exception handler, they are different. The value that is set by BOOM is off by 2. According to the designer, the bug is due to the handling of compressed RISC-V instructions (RVC). Specifically, handling of exceptions on misaligned instructions appeared to be broken. The bug

disappears in later commits of the BOOM.

6.2.3 Dromajo+Logic Fuzzer Evaluation

Table 6.4 summarizes the bugs exposed by merely integrating Dromajo into the design of open-source cores. Below is the detailed explanation of each bug.

Bug ID	Core	Short description
B10	CVA6	incorrect trap cause
B11	CVA6	arbiter locks with gnt 0
B12	BlackParrot	core hangs on access to irregular memory region
B13	BlackParrot	backend backpressure breaks instruction ordering

Table 6.4: Summary of the bugs exposed in three RISC-V cores by Logic Fuzzer enhanced Dromajo.

Bug ID 10. This bug surfaces out when we mutate the ITLB entries. The random mutation made the instruction TLB entry valid but the corresponding translation was mutated to a non-existent memory region. As a result, both Dromajo and CVA6 threw an exception and steered the execution flow to the exception handler. Dromajo halted the execution when reading an *mcause* register within the handler due to the mismatch. When trapping, Dromajo correctly sets the *mcause* value to 1 which indicates the exception cause *Instruction Access Fault*. CVA6, on the other hand, incorrectly set the value to 12 which indicates *Instruction Page Fault*. According to the designer, this is because CVA6 implementation aliases the access and page faults in the instruction front-end and treats everything as instruction page faults.

Bug ID 11. This bug is exposed when we create artificial backpressure at the FIFO's full signal in the cache subsystem and results in the complete hang of the

system. The purpose of this FIFO is to queue memory requests that are coming from the icache. The full signal, in turn, is used to form a request logic for the arbiter, which performs arbitration between icache and dcache requests. The randomized backpressure stirs up the arbiter's states and locks the grant signal indefinitely at 0, not allowing any of the requests to go down.

Bug ID B12. The Black-Parrot microarchitecture defines a FIFO queue between the frontend and the backend of the core. The purpose of this FIFO is to enqueue specific commands, such as PC redirect and state reset, from the backend to the frontend. This bug is exposed when we insert the congestor at the FIFO's ready signal. We create artificial backpressure by randomly pulling this FIFO's ready signal low. When we run the tests with the inserted congestor, Dromajo catches the mismatch as BlackParrot starts committing instructions with the wrong PC. According to the designer, BlackParrot's backend cannot handle the backpressure. Because the microarchitecture has no stalling points past the decode stage, some backend commands will be lost if the queue is not ready.

Bug ID B13. This bug is exposed when we generate irregular addresses from BlackParrot's Branch Target Buffer (BTB). The BTB fuzzing generated the address that maps to off-chip memory. This scenario resulted in the complete freeze of the system. According to the designer, addresses that are routed to a tile must be responded to. However, BlackParrot decoded the address and routed it to a specific device on the tile, such as cfg or clint. In the case when no device matched, it hanged.

6.2.4 Dromajo+Marionette Models Evaluation

CVA6's Branch Predictor Marionette. Figure 6.1 illustrates the misprediction count of various branch predictor Marionette Models along with the actual branch predictor of CVA6. We instantiated 16 Marionette Models, each configured with different branch history table size, starting from 16 entry table and gradually increasing the size up to 512K entries. We then boot the Linux and observed the behaviour the models simultaneously in a single run. The results of this experiment show that the Marionette Model with the configuration that matches the CVA6 branch predictor, 128 entries, had no deviations. The plots for these two branch predictors, blue and red dotted lines, completely overlap in Figure 6.1. Running simulation with 16 branch predictor Marionette Models introduced 2% run-time overhead.

BOOM's Shared Memory System Mariontte. Listing 6.1 is the C code that we compiled into RISC-V binary and ran on the multi-core configuration of BOOM. The program runs in machine mode and operates on shared data. Despite being simple, it heavily exercises the memory subsystem components. The program generates a constant flow of loads and stores to the same cache-line activating logic responsible for cache coherence and memory consistency.

Simulating these kinds of programs in a multi-core setting **with co-simulation enabled was NOT possible prior to this work** due to the reasons explained in Section 4.5. In fact, in our experiments, the co-simulation reported a false positive mismatch on the load instruction several seconds into the simulation. After implementing and de-

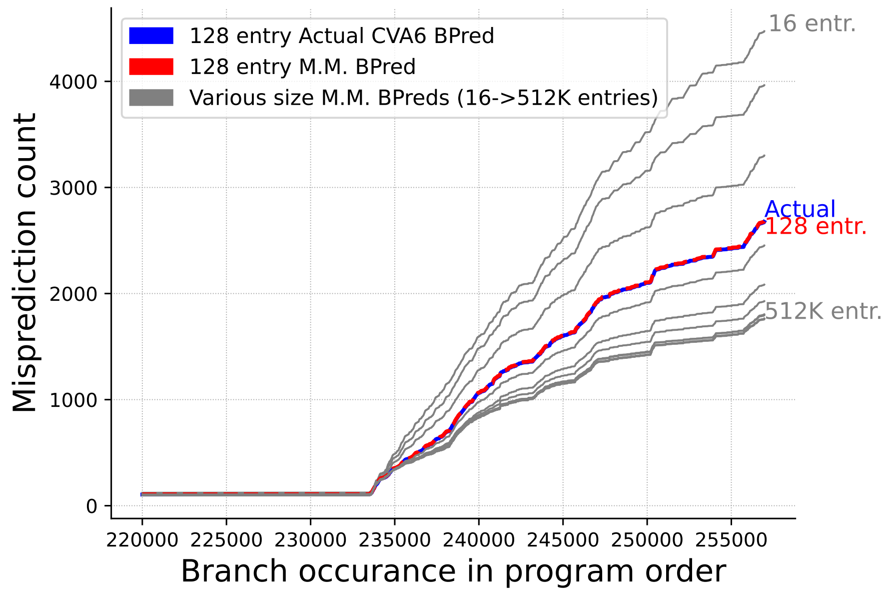

Figure 6.1: Number of branch misprediction of 16 different branch predictor Marionette Models along with the branch predictor of CVA6. The simulation is running fraction of the Linux Boot. The Marionette Model with the same BHT size as CVA6's branch predictor behaves identically.

```
static uint32_t data[8] = {0, 0, 0, 0, 0, 0, 0, 0};
int main() {
    while (true) {
        for (uint32_t i = 0; i < 8; i++) {
            ++data[i];
            if (data[i] > 100) {
                data[i] -= 100;
            }
        }
    }
}
```

Listing 6.1: C code to exercise shared memory system.

ploying Marionette Model into BOOM's verification infrastructure, we co-simulated the above program until the simulation time-out of 10M cycles. Figure 6.2 is the visual snapshot of the simulation that illustrates how memory operations go through the BOOM's

pipeline over time and pull the strings of the Memory System's Marionette.

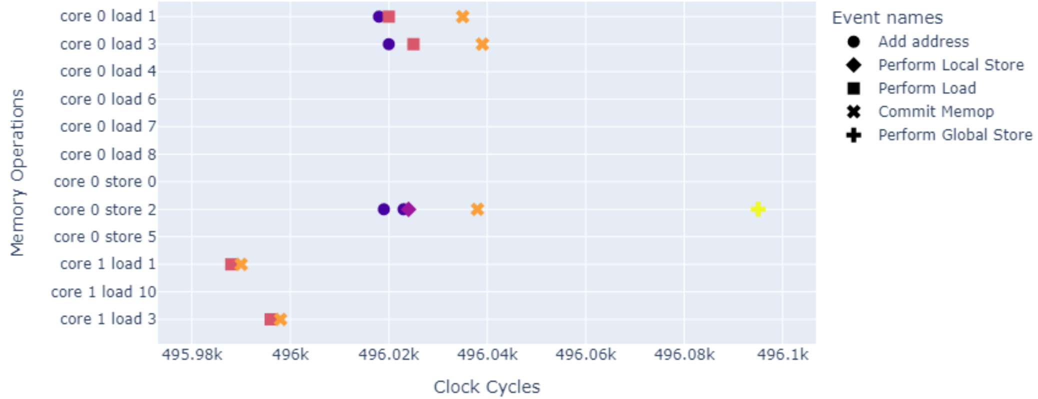

Figure 6.2: Timing diagram of memory operations going through the dual-core BOOM pipeline and pulling strings of the Marionette.

6.2.5 Dromajo+Marionette Model+Logic Fuzzer Evaluation

Enabling co-simulation in multi-core settings opened up a whole new verification space. This verification space has a place for the application of Logic Fuzzer. In other words, all of the tools and the methodology that I described in this dissertation can be used together for more effective and productive verification.

In this section, I evaluate how Logic Fuzzer allows more rigorous testing of memory operation event ordering. For this evaluation, I define the following terms:

- Observation window - a range of consequent events under observation.
- Observation window width - a scalar number that defines the size of the observation window. In other words, it defines how many events are under observation.

- Event ordering - an ordering of the events in the observation window.

To quantify how rigorously the event ordering is exercised during the simulation, we define an observation window of size N and slide it through the whole simulation. As we slide, we count the number of unique event sequence occurrence in the simulation.

The reasoning behind event ordering observation is that different ordering can potentially bring the shared memory system into a different microarchitectural state. For instance, consider the case when two cores are each processing a store instruction to the same address. The L1 cache-line states of the cores will differ depending on which core globally performs the store first. If core 0 globally performs the store first, core 0's corresponding L1 cache-line will be in M state, and core 1's L1 cache line gets invalidated. If, on the other hand, core 1 globally performs the store first, the opposite of the described state will be valid.

To understand the definitions more clearly, let's apply the terminology to the toy example illustrated in Figure 6.3. For clarity purposes, the illustration in the Figure limits the number of events to five, and letters A to C annotate each event. To be precise, "A" annotates the addition of the address for load instructions from core 1. "B" annotates the performance of the load from core 1. Finally, "C" annotates the addition of the address for store instruction from core 0.

Next, let's define an observation window size to be two. Sliding this observation window through the illustrated example simulation period yields the following sequences: AB, BA, AB, and BE. Three (out of four) of the observed orderings are unique.

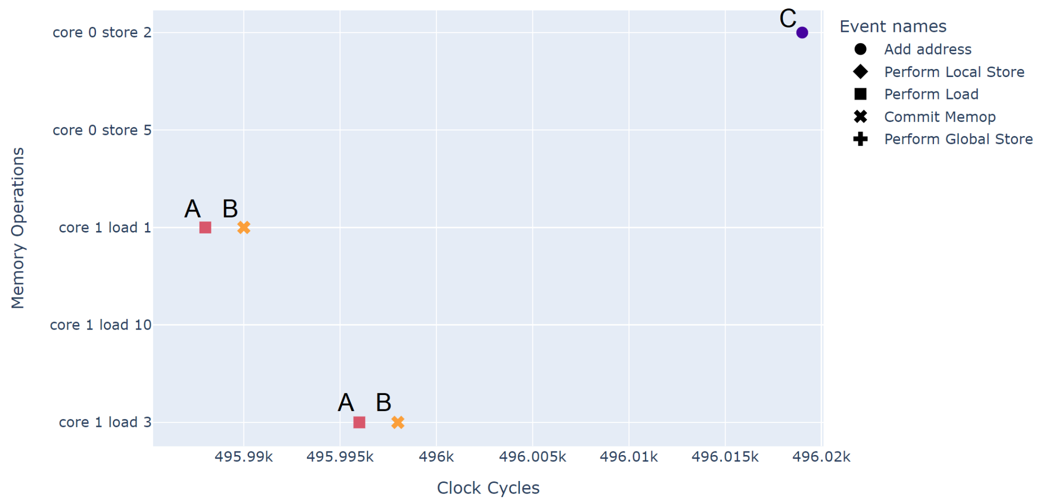

Figure 6.3: Zoomed-in timing diagram with annotations of memory operations going through the dual-core BOOM pipeline.

For the experiment, I inserted a simple congestor (Section 5.2.1) in the load-store unit of both cores at the "full" signal of the load queue. We then run the simulation twice with LF and without LF. For each run, we kept track of the number of unique event orderings with the observation window size of 8. Figure 6.4 shows the dynamics of the unique event ordering appearance during two runs. The Logic Fuzzer caused the discovery of the unique events orderings both faster and in greater amounts (36% more after committing 1M instructions).

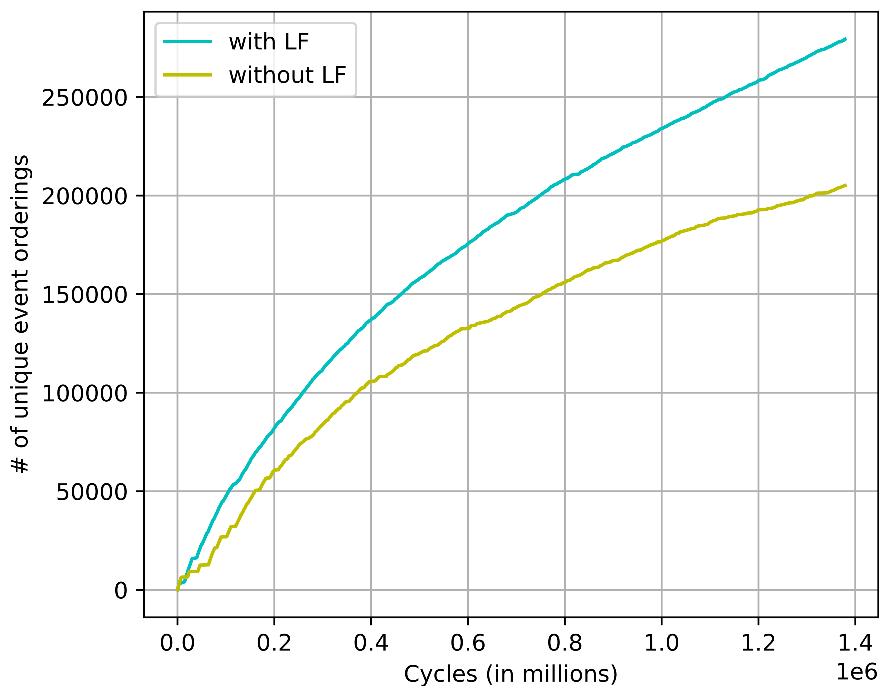

Figure 6.4: The dynamics of unique event ordering occurrence during the simulation with and without Logic Fuzzer.

6.3 Discussion

6.3.1 Bug Hunting with LF and False Positives

The co-simulation mismatch is what tells if the bug was manifested or not. We start debugging only if the activity created by LF propagates to the architectural state and gets flagged by Dromajo. The process of proving or disproving if the mismatch is an actual bug is no different from regular RTL debugging. We then discuss the debugging results with the designer who confirms or rejects the validity of the finding.

Logic Fuzzer had two false bugs (not presented in the dissertation) – one in CVA6 and one in BOOM. Let us mention that traditional verification practices have the same issue of false positives. The fact that we find bugs in an automatic way does not qualitatively affect this issue, only quantitatively.

6.3.2 Toggle Coverage

The signal is said to be *toggled* if its value switched $0 \rightarrow 1$ and $1 \rightarrow 0$ at least once while executing the test. Toggle coverage is one of the proxy metrics that is used both in industry [23, 45] and academia [28] to gain confidence about the correctness of the design-under-test.

Figure 6.5 illustrates how the toggle coverage increases as we run the verification binaries. Logic Fuzzer increased the toggle coverage on average by 1%.

Although increased coverage is a beneficial side-effect of fuzzing, **we want to emphasize that increasing coverage in and of itself is NOT the purpose of the Logic Fuzzer. The purpose is to create an irregular execution flow, which, most of the time, is not captured by the coverage metrics**. Toggle and similar types of code coverage are not full indicators that the system is verified. They are just proxy metrics.

For example, from the bugs found by the Logic Fuzzer, only one (B12) is directly correlated to the toggle coverage. The remaining three bugs do not correlate with toggle coverage. The bugs were detected because of the randomized events created by LF. It was a combination of signals and states that even with a 100% toggle coverage

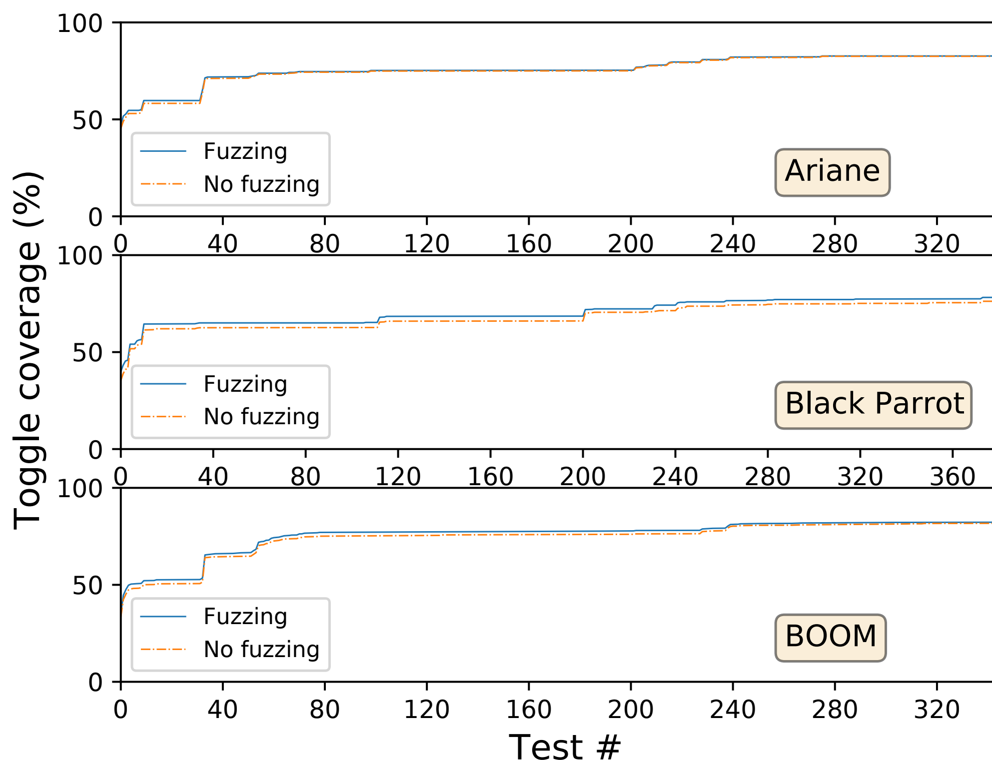

Figure 6.5: Coverage increase when running verification binaries

may not have been detected.

6.3.3 Marionette Model Integration

In the experiments, the identification of the correct conditions was the most challenging part. This was primarily due to the steep learning curve in familiarity with the RTL code base. We are certain that the effort of this step reduces to a minimum if the verification engineer could closely collaborate with the designer, which is usually the case in industry.

Also, it is worth noting that once the correct conditions are identified the inte-

gration of the marionette model into the infrastructure could be completely automated. In our experiments, for productivity purposes, we developed a python utility script that automatically integrates the marionette into the test-bench. The script accepts a JSON file with the $condition \rightarrow API$ mapping information, similar to what is presented in Table 4.1. The script then generates required DPI wrappers, Verilog black boxes and places the generated files to the proper directory locations. For instance, with this utility script, it took around two development-hours to integrate Marionette Model into newly generated quad-core BOOM configuration with L2 cache.

Chapter 7

Conclusion and Future Opportunities

> Don't be satisfied with stories, how things have gone with others. Unfold your own myth.
>
> — Rumi

7.1 Conclusion

In this thesis, I have examined some of the problems in current microprocessor verification methodologies and describe some of my efforts to address these problems. I presented Logic Fuzzer, a methodology that brings the processor execution outside its normal flow by randomizing the microarchitectural state at the places that do not affect functionality. We presented several Logic Fuzzer variants and showed that it could uncover additional bugs in the simulation phase by creating atypical scenarios without the generation of additional tests.

I illustrated the effectiveness of Dromajo by exposing nine bugs in CVA6, BlackParrot, and BOOM. The enhancement of Dromajo with Logic Fuzzer exposes additional two bugs in CVA6 and two bugs in BlackParrot. Logic Fuzzer was not able to find additional bugs in BOOM. However, we demonstrated the applicability and effortless integration of the tools into existing testbench environments. Logic Fuzzer found difficult bugs. In three cases, the bugs were not directly correlated with toggle coverage, which means that the tests exercised the associated logic. The bugs were still there because a combination of events was needed to reach the bug condition. Logic Fuzzer exposes these bugs.

Finally, I addressed the fundamental issue of the co-simulation technique that limited its applicability to single core simulations only. I presented a novel methodology, Marionette Models, which enabled the co-simulation of systems in multi-core settings with shared memory.

I expect that Dromajo will enhance the quality of existing RISC-V cores. Furthermore, I believe Logic Fuzzer and Marionette Modeling techniques will be applied beyond RISC-V cores and be beneficial to a broader computer architecture community.

7.2 Future Work

I am currently investigating several ways of improving the effectiveness of Logic Fuzzer. To be specific, the items that could further improve LF include the identification and testing of other *fuzzable* logics in the microprocessor, such as reordering of

outstanding memory requests and randomization of fixed priority muxes and arbiters. We are also looking into training a machine learning model to orchestrate the inserted Logic Fuzzers in the system.

I believe that Dromajo and Logic Fuzzer are great tools to help the RISC-V ecosystem have a more robust infrastructure. Besides the presented contributions, this work opens up new research opportunities and could be extended further in new areas. To mention a few, (1) Logic Fuzzer can be integrated with tools like Coppelia [55]. Coppelia uses symbolic execution techniques to generate exploits that will bring the processor into a vulnerable state. Although this work is proposed in the context of hardware security, I believe it can be extended towards functional verification. We could potentially provide Coppelia with the failing case exposed by the Logic Fuzzer and let it try to generate the code. The successful generation of the exploit will prove the presence of the real bug. (2) Integrating Logic Fuzzer in the fully elastic systems [37] could expose even more bugs than in non-elastic designs. (3) The RISC-V checkpoints created by Dromajo could also be leveraged for works that do statistical sampling for performance and power.

Marionette Models have several vectors for further development and improvement. The first task that is on my radar is more rigorous testing of the shared memory system's Marionette Model by running programs with atomic instructions, litmus tests and eventually running Linux. The second task is to implement more advanced branch predictor models such as TAGE and evaluate the performance of BOOM's branch predictor.

One of the issues both with Logic Fuzzer and Marionette Models is the overhead of the integration. Even though the overhead is already small, I envision an extremely user-friendly integration that reduces the overhead to a minimum. I envision software that accepts an arbitrary RTL design as an input and has a rich library of Marionette Models and Logic Fuzzers that are drag-dropped to the design.

I believe this dissertation was a solid step toward more effective and productive microprocessor verification. I finish this dissertation with a positive mindset and with the vision that we, as a community, will be able to develop efficient tools which will catch all bugs before fabrication and eliminate the need for re-spins.

Bibliography

[1] [information is disclosed per double-blind peer review guidelines].

[2] W. Anderson. Logical verification of the nvax cpu chip design. In *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers Processors*, pages 306–309, 1992.

[3] Azad, Zahra; Delshadtehrani, Leila; Zhou, Boyou; Joshi, Ajay; Gilani, Farzam; Lim, Katie; Petrisko, Daniel; Jung, Tommy; Wyse, Mark; Guarino, Tavio; Veluri, Bandhav; Wang, Yongqin; Oskin, Mark; Taylor, Michael. The blackparrot processor: An open-source industrial-strength RV64G multicore processor. Mar 2019.

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanov. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012.

[5] Brian Bailey, Harry Foster, Chirag Gandhi, Tom Anderson, Craig Shirley, Adam Sherer, Rajesh Ramanujam, Roger Sabbaugh, Vigyan Singhal, and Kevin McDermott. When bugs escape), 2018.

[6] Chipyard. 8.2. communicating with the dut.

[7] Shenghsun Cho, Mrunal Patel, Han Chen, Michael Ferdman, and Peter Milder. A full-system vm-hdl co-simulation framework for servers with pcie-connected fpgas. FPGA '18, pages 87–96, New York, NY, USA, 2018. Association for Computing Machinery.

[8] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP), Aug 2017.

[9] M. Chupilko, A. Kamkin, A. Kotsynyak, A. Protsenko, S. Smolov, and A. Tatarnikov. Test program generator microtesk for risc-v. In *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, pages 6–11, 2018.

[10] Western Digital. Whisper instruction set simulator.

[11] Schuyler Eldridge, Alper Buyuktosunoglu, and Pradip Bose. Chiffre: A configurable hardware fault injection framework for risc-v systems. In *2nd Workshop on Computer Architecture Research with RISC-V (CARRV '18)*, 2018.

[12] RISC-V Foundation.

[13] RISC-V Foundation, Jul 2013.

[14] Google. Sv/uvm based instruction generator for risc-v processor verification, Jan 2019.

[15] Mentor Harry Foster. Part 12: The 2018 wilson research group functional verification study (ic/asic verification results trends), 2019.

[16] Mentor Harry Foster. Part 7: The 2018 wilson research group functional verification study (ic/asic design trends), 2019.

[17] Mentor Harry Foster. Part 8: The 2018 wilson research group functional verification study (ic/asic resource trends), 2019.

[18] Vladimir Herdt, Daniel Grosse, Eyck Jentzsch, and Rolf Drechsler. Efficient cross-level testing for processor verification: A risc-v case-study. In *Forum on Specification and Design Languages (FDL)*, 2020.

[19] R. C. Ho, C. Han Yang, M. A. Horowitz, and D. L. Dill. Architecture validation for processors. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 404–413, 1995.

[20] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. Instruction-level abstraction (ila): A uniform specification for system-on-chip (soc) verification. *ACM Trans. Des. Autom. Electron. Syst.*, 24(1), dec 2018.

[21] Intel. Understanding write combining on arm, Nov 1998.

[22] 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), pages 209–216, Nov 2017.

[23] Jing-Yang Jou and Chien-Nan Liu. Coverage analysis techniques for hdl design validation. *IEEE Asia Pacific Conference on Chip Design Languages*, 01 1999.

[24] Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. Effective processor verification with logic fuzzer enhanced co-simulation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 667–678, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodov, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing testing with formal verification in intel coretm i7 processor execution engine validation. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 414–429, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[26] M. Kantrowitz and L. M. Noack. I'm done simulating; now what? verification coverage analysis and correctness checking of the decchip 21164 alpha microprocessor. In *33rd Design Automation Conference Proceedings, 1996*, pages 325–330, 1996.

[27] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA '98*, 1998.

[28] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. RFUZZ: coverage-directed fuzz testing of RTL on fpgas. In Iris Bahar, editor, *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2018, San Diego, CA, USA, November 05-08, 2018*, page 28. ACM, 2018.

[29] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[30] Lee and Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, 1984.

[31] lowRISC. Ibex core verification, Aug 2019.

[32] Imperas Software Ltd. Imperas announce first reference model with uvm encapsulation for risc-v verification.

[33] Imperas Software Ltd. Ovp risc-v solutions.

[34] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Pipecheck: Specifying and verifying microarchitectural enforcement of memory consistency models. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 635–646, 2014.

[35] M. Naylor and S. Moore. A generic synthesisable test bench. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 128–137, 2015.

[36] OpenHWGroup, July 2020.

[37] R. T. Possignolo, E. Ebrahimi, H. Skinner, and J. Renau. Fluid pipelines: Elastic circuitry meets out-of-order execution. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 233–240, 2016.

[38] Anmol Sahoo, Dec 2019.

[39] Pavel Shamis. Understanding write combining on arm, Nov 2020.

[40] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *ACM SIGPLAN Notices*, 37, 09 2002.

[41] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. volume 31, pages 336– 347, 07 2003.

[42] Christopher Batten Shunning Jiang, Christopher Torng. An open-source python-based hardware generation, simulation, and verification framework. In *Proceedings of the First Workshop on Open-Source EDA Technology (WOSET18)*, 2018.

[43] Pradeep S. Sindhu, Jean Marc Frailong, and Michel Cekleov. Formal specification of memory models. 1992.

[44] Wilson Snyder. Verilator.

[45] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design Test of Computers*, 18(4):36–45, 2001.

[46] Esperanto Technologies. Dromajo - esperanto technology's risc-v reference model, Dec 2019.

[47] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software, 2021.

[48] UCSC. Marionette models, Nov 2022.

[49] I. Wagner, V. Bertacco, and T. Austin. Stresstest: an automatic approach to test generation via activity monitors. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 783–788, 2005.

[50] Sheng-Hong Wang, Rafael Trapani Possignolo, Qian Chen, Rohan Ganpati, and Jose Renau. LGraph: a unified data model and api for productive open-source hardware design. In *Open-Source EDA Technology, Proceedings of the Second Workshop on*, WOSET'19, November 2019.

[51] Claire Wolf. Symbiyosys (sby) - front-end for yosys-based formal verification flows, Jun 2017.

[52] D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a multiprocessor cache controller using random test generation. *IEEE Design Test of Computers*, 7(4):13–25, 1990.

[53] Alberto Dassatti Xavier Ruppen, Roberto Rigamonti. Test program generation for functional verification of powepc processors in ibm. In *Barcelona RISC-V Workshop*, 2018.

[54] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit RISC-V core in 22-nm FDSOI technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.

[55] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. End-to-end automated exploit generation for validating the security of processor designs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2018.

[56] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. May 2020.