A Research-Fertile Co-Emulation Framework for RISC-V Processor Verification

Anoop Mysore Nataraja

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2023

Committee:
Michael Taylor, Chair
Scott Hauck

Program Authorized to Offer Degree:
Department of Electrical and Computer Engineering

University of Washington

Abstract

A Research-Fertile Co-Emulation Framework for
RISC-V Processor Verification

Anoop Mysore Nataraja

Chair of the Supervisory Committee:
Michael Taylor
Electrical and Computer Engineering

As processor design complexities increase, so do their verification complexities. As a consequence, processor verification has slowed down and become less reliable. The recent drift towards agile chip design philosophies and increasingly expensive ramifications of bugs and security vulnerabilities only aggravate the situation. Despite advancements in expensive commercial verification solutions, there is still a need for cost-effective, fast and high-confidence open-source verification solutions. Automated verification methodologies have emerged as promising candidates for their speed and reliability; however, automation comes with its fair share of open problems – which an inexpensive, easy-to-setup, and modifiable experimentation platform can help research.

This thesis presents an open-source framework, ZP Cosim, for FPGA-accelerated cosimulation of RISC-V processors. The framework is cost-effective, customizable, and scalable to FPGA-clusters, and has been field-tested against the silicon-validated BlackParrot processor. The framework additionally offers a novel implementation of automated coverage instrumentation and a customizable FPGA shell for coverage and trace extraction. ZP Cosim achieves a speedup of over 2000x against cosimulation in a popular RTL simulator. The application of the framework to BlackParrot resulted in the discovery of 4 designer-acknowledged microarchitectural bugs. The thesis discusses these in detail along with observations of the

coverage effected by popular benchmarks and randomly generated programs.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am incredibly grateful to the Bespoke Silicon Group, home to some of the world's most obsessive, high-functioning, and accomplished chip-designers. Their tireless drive, constant support, and visionary ideas have been instrumental in enabling the completion of this thesis. I feel exceptionally fortunate to have had the opportunity to work alongside the brilliant minds composing the group.

I would like to express my heartfelt thanks to my dynamic advisor, Prof. Michael Taylor, and my incredibly knowledgeable mentor, Daniel Petrisko. Their insatiable thirst for knowledge, their lead-by-example work ethic and their unwavering encouragement have been a constant source of motivation. Without their timely insights and the fruitful collaborations Michael facilitated, I would have had to expend a lot more time and effort for far less valuable outcomes. I believe humans learn best by mimicking other (fascinating) humans. To me, they are Michael, Daniel, and Madhav (at the Indian Institute of Science) – I hold great respect for each of them and I dream of surpassing them some day. I cannot thank them enough.

I'm also especially thankful to:

- Prof. Jeff Bilmes and Prof. David Kohlbrenner for two of the best courses I've had on Statistical Learning, and Hardware Security, respectively,
- Prof. Scott Hauck, for his valuable comments on this thesis, and
- My loving parents and my sister, for probably the only external constancy in my life.

Finally, I'm thankful to the $\{139 + 1\}$ too-many-to-name friends, mentors, and acquaintances for everything nice they enabled in my life.

Portions of this work were partially supported by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement numbers FA8650-18-2-7863 and FA8650-18-2-7856; NSF grants SaTC-1563767, SaTC-1565446. This work intersects and leverages research and infrastructure created by the members of the Bespoke Silicon Group, spanning across accelerators ([28, 45, 136, 132, 90, 88, 139, 24, 127, 98, 51, 101, 20, 99, 50, 52, 75, 126]), ASIC Clouds ([121, 133, 118, 108, 76, 77, 82, 110]), open source hardware ([120, 111, 42]) RISC-V ([91, 97, 96, 39, 138, 16, 125, 70, 94, 84, 35]), Network-on-Chips ([71, 92, 141, 115, 78]), security ([32, 58, 59, 17]), benchmark suites ([123, 79, 22]), dark silicon ([107, 106, 107, 109, 26, 53, 50, 126]), multicore ([91, 56, 54, 55, 105, 115, 117, 112, 104, 78, 113, 116, 114, 128]), compiler tools ([67, 46, 65, 48, 66, 64, 47, 135, 15]) and FPGAs ([140, 62, 22]).

DEDICATION

To Suhani; for the strange happiness she was responsible for.

Chapter 1

INTRODUCTION

Increased processor design complexity is generally an acceptable tradeoff for better performance and energy efficiencies. This complexity is often a result of high-degree of hardware customization through host coupled accelerators, coprocessors, complex memory hierarchies and interfaces, and nuanced microarchitectural innovations. Consequently, verification of complex processor hardware is significantly challenged in timespan, complexity, and reliability. In addition, with the recent fascination with agile chip design philosophies [68], aggressive time-to-market windows, aggravating ramifications of bug-fixes post-silicon and resulting expensive ASIC respins, a fast and high-confidence verification becomes a fundamental necessity for commercial chip design.

Present-day processor designs easily span millions to billions of logic gates leading to a humanly untamable, or practically inexhaustible, hardware state-space[1] to verify, and because of the nuances of the customized microarchitectures and the runtime ecosystems, there is often a high non-recurring engineering (NRE) cost to setting up and exercising processor verification. In practice, however, a significant fraction of the state space is safely ignorable, never exercisable, or unimportant during verification for various reasons. In fact, some fraction of the theoretically possible state-space of complex processors will likely never be exercised in the entirety of the product lifetime [2]. Moreover, it may often be crucial to test nuanced microarchitectural innovations manually because complex states and state-

[1]Hardware state-space in the context of verification is the set of hardware states and/or functionalities (that need to be tested and validated).

[2]Consider for example, a processor design with 1000 control path expressions (equivalent to decision statements in the design description) which is not uncommon in a modern-day multicore. The resulting hardware state-space is 2^{1000}. Assuming that the processor runs at 5 GHz continuously for 10 years without ever repeating a state, the state-space it would cover would be less than 2^{61}

transitions may not lend themselves to randomized test vectors[3]. Nevertheless, it remains theoretically possible to test the design reliably by maneuvering the structural design description automatically – presumably, through well-devised coverage metrics that prioritize the annotation of important states and state-transition in the design followed by intelligent test-vector mutations.

A concise hardware state-space enables effective hardware fuzzing[4] over a chosen constraint – usually, program length (or equivalently, simulation time). Critical feedback, often in the form of functional or structural coverage[5], assists the fuzzer in reaching hitherto uncovered (or unexercised) states, usually via learnable test vector mutations. However, manually identifying reliable and useful coverpoints to infer coverage is laborious. This motivates automated coverage metrics. Complementing fuzzing with cosimulation against a golden reference model of the hardware obviates needing to define and reason about different states of the hardware manually or formally, thus shorting high-confidence with fast and reliable verification. Ultimately, the most important measure of a hardware verification infrastructure is the number of bugs discovered (new and inclusive of prior discoveries), and the ideal verification infrastructure should excel at this.

With this in mind, the rest of the thesis presents ZP Cosim, a framework for fuzzing and hardware verification of complex RISC-V processors, while remaining faithful to the following adjectives:

Research-Fertile ZP Cosim allows accelerated critical evaluation of long running benchmarks and the coverage they effect on the hardware. The insights gained through running SPEC benchmarks on BlackParrot processor are discussed in Chapter 4. Moreover, with the recent boom in hardware fuzzing based research [72, 124, 130], a ZP

[3]Randomized test vectors are one of the alternatives to manually constructing test vectors for testing specialized states of the hardware.

[4]Hardware fuzzing is an iterative exercise of deploying random test vectors on the design-under-test for converging an objective – usually, coverage.

[5]Coverage is discussed in Section 2.2.

Cosim's fast, cheap and open-source characteristics can accelerate fuzzing and security research.

Cost-Effective ZP Cosim undercuts large scale verification costs which are often prohibitive because of commercial tool licensing, expenses related to server-time acquisitions, and non-recurring engineering costs, by being completely free to use, needing only modest $300 FPGA boards, and being fairly portable.

Fast ZP Cosim is FPGA-accelerated and farm-deployable[6] on small to medium-scale FPGA clusters with independently operable verification routines.

Portable ZP Cosim allows for easy swap-in of other RISC-V implementation with reasonably minimal modifications to the hardware descriptions. In addition, because ZP Cosim's coverage automation is based on Verilog/SystemVerilog which is a widely-adopted hardware description language (HDL) and what most high-level HDLs lower to, and because the instrumentation is non-intrusive and places no requirements on the design description whatsoever, the infrastructure provides good portability.

Customizable Because ZP Cosim is open-source, it is completely customizable and integrable with other tools in the open-source ecosystem for an expansive feature set. It can be specialized according to individual needs and developed upon for custom testing requirements. This is an important quality because many commercial solutions that offer comparable feature set as ZP Cosim, often are limited in customizability.

Field-Tested ZP Cosim has been exercised on the silicon-validated, previously extensively tested BlackParrot processor revealing 4 new and important bugs.

---

[6]On FPGA farms or FPGA-clusters that can be hooked up to a Local Area Network and operated through a host machine.

Thesis Organization

Chapter 2 discusses popular concepts in processor verification and builds an outline of an ideal, self-contained verification infrastructure. Chapter 3 describes the pith of this thesis – an implementational baby-step towards a self-contained verification infrastructure. Chapter 4 discusses metrics to evaluate ZP Cosim, the insights gained as a result, and describes critical bugs discovered through the exercising of ZP Cosim on BlackParrot processor. The final chapter concludes the thesis with key takeaways.

Chapter 2

LANDSCAPE OF SYSTEM-ON-CHIP VERIFICATION

Verification of processors is a crucial exercise for reducing the aggregate costs of chip design by lowering the probabilities of microarchitectural and security bugs which helps avoid expensive ASIC respins and product recalls later in the product cycle. In fact, an industry-wide study conducted by Harry Foster, shows that over 70% of a variety of chip designs needed ASIC respins because of predominantly design-related defects[44] that a well-implemented verification plan could theoretically mitigate. Additionally, the insights gained during verification and performance modelling is often invaluable to future design iterations – both with improvements to the design, and with avoiding costly pitfalls and bad design practices in the future.

This thesis evaluates 3 key aspects of verification:

- Speed of verification, often under the constraints of a given time budget,
- Completeness of verification, or, the tendency to discover bugs and security vulnerabilities, and,
- Cost of verification.

Processor verification is a balance between many factors such as time-to-market and New Product Introduction (NPI) windows, design and verification budgets – both in people-years, and in dollars for the tools and infrastructure costs, accessibility and modifiability of the tools and infrastructures according to specialized needs, and the reasonableness of effecting and distributing software patches. A 2022 research study on a mix of commercial ASIC verification projects by the Wilson Research Group [1] gives us useful insight on the

landscape of these tradeoffs: over 66% of the surveyed mix of ASIC projects are behind schedules, and over 60% of the project lifetime is spent in verification.

However, with increasingly modular designs, practices of design reuse, and highly structured electronic design automation tools, verification has become highly conducive to automation. Despite long-standing practices of automation in verification, the process is still fundamentally human-in-the-loop.

Empirically, chip designers have relied on software simulations, emulations/FPGA-prototyping, and formal verification to satisfy various verification goals. And in each practice, there are many people-hours invested in the infrastructure setup, exercise, debug, evaluation, and bug fixes.

However, with growing complexity of designs, software simulations are inevitably getting slower, although the increasing quality and reliability of cycle-accurate RTL simulators such as Synopsys VCS, and functional modelers such as Verilator [100] maintain the simplicity of setting up software simulations. Formal verification is getting harder and more time-consuming to set up for complex systems and more complex specifications, and are often limited to localized modular verifications or security evaluations, despite their reliability. Modern practices notoriously under-formalize hardware in the design-time which makes setting up formal models time-consuming and laborious in verification-time. And emulating designs, has the advantage of order of magnitude of speedups over simulations, and, with the improving FPGA synthesis and placement tools, continue to remain simple and efficient in limited number of cases. Of course, with FPGA emulations, the test-time debuggability is severely limited and often involves complex test harnesses and scaffolding in the original design for even the most minimal debuggability.

In practice, however, commercial-grade verification methodologies rely on a combination of human intelligence with each of these verification composites at various developmental stages of the chip design, and use a variety of commercial ASIC verification tools and tool-experts for achieving final verification sign-offs.

2.1 Faster Verification

Fast verification leads to early fixing of bugs and security vulnerabilities, and in time-constrained cases, discovery of more bugs. Two major enablers of faster processor verification are design-reuse and design modularity which are embraced by many commercial and open-source processor designs to varying extents. BaseJump STL [119] is one of the earliest SystemVerilog standard template libraries (STL) [4, 2] popularized in the open-source community that promotes a high degree of design modularity. The use of such parameterized, pre-verified, reliable templates obviates the need to verify template-internals every time the functionality is implemented in a larger design.

Standardized IO interfaces, and decoupled module interfaces are integral parts of such STLs. Standard interfaces enable reuse of verification infrastructures, reducing or sometimes eliminating the time needed to setup and adapt new or existing simulation, emulation, and testing infrastructures. Additionally, raising the abstraction level in which designers describe hardware, such as with domain-specific languages (DSLs) like Chisel[23], TL Verilog[61], Bluespec[27], etc., also raises the abstraction of verification. Similarly, with the use of hardware generator infrastructures such as the RocketChip (part of Chipyard)[18], furthers the philosophy of design (and verification infrastructure) reuse and composability, and with high-quality, verified generators, the verification time can be reduced manifold.

Despite such practices, verification is still the predominant phase of chip design. The key insight here is that even with high degrees of modularity and design reuse that enable larger and more complex designs, the integrations and interfacial behaviors of the composite modules and hardware IPs remains to be verified. Even considering just the extra-modular design elements is a high-enough hardware state-space for exhaustive hardware verification. However, because of the aforementioned nature of hardware design, much of this state-space is an unintentional consequence of modularity and often, it is hidden behind parameterizations and functional redundancies. Understandably, such instances lend themselves better to being recognized and tested manually, thereafter increasing human involvement in veri-

fication iterations – which slows-down verification and reduces reliability. In most cases, a randomized application of test vectors to the hardware interfaces, referred to as fuzzing, helps in exercising much of the typical state-space, and the specialized states are manually tested and accounted for. At such a juncture, the property of coverage of verification, discussed in Section 2.2, becomes important.

2.1.1 FPGA-Accelerated Verification

One way of naively increasing speed of verification is to invest in servers for large scale parallel simulations of designs. Commercial solutions like this include the Synopsys Zebu Server 5[102]. Another possibility is moving to cloud-FPGA emulations [73] instead of large scale simulations, with higher initial setup time that can be easily amortized over many verification iterations. Commercial solutions such as the Cadence Palladium Z1/Z2 and Proteum X1/X2 [31, 30] provide many different kinds of acceleration and support larger design emulation and software bringup. Each of the options come with their own drawbacks of being expensive and non-pliable.

2.2 More Complete Verification

Completeness of verification is most commonly measured with the *Coverage* metric. Coverage is an important quality of verification and is defined as an approximate indication of the extent to which a round of testing satisfies a set of predefined verification objectives – usually, the completeness or the effectiveness of the round of testing in exercising the hardware.

Coverage in hardware designs is broadly categorized as structural or functional. Structural coverage assesses coverage over structural elements of the design – usually, as described with a hardware description language, and as such, can be automated in practice. Functional coverage, on the other hand, assesses coverage over (usually) higher-level functional behaviors of the design and as such, functional coverage, at times, relies on manual specification which

involves high engineering effort, although, certain metrics like FSM coverage[1] do a reasonably good job in assessing functional coverage in well structured hardware descriptions.

The abstractions in which coverage is assessed can vary:

- Architectural or ISA-level coverage usually assesses coverage over instruction mixes, operand combinations, ISA-defined CSRs. These can be evaluated with functional ISA simulators.

- Microarchitectural coverage, on the other hand, assesses coverage over implemented hardware entities. ISA-level coverage is, generally, a subset of microarchitectural coverage, assuming all ISA-defined abstract elements are physically implemented in the microarchitecture. Of course, the chosen coverage metric, can sometimes identify no more coverpoints in the microarchitecture than defined by the ISA, such as with the CSR coverage metric that only identifies CSR registers in the mircroarchitecture as coverpoints.

ZP Cosim's coverage metric assesses microarchitectural coverage as it scans the microarchitecture description (hardware) for identifying coverpoints. Because of microarchitectural coverage's wider statespace, there is a better chance of finding bugs.

Other aspects of coverage such as unreachability, toggle-only coverage, etc., are presented in section 3.1.7. A key point to remember is that coverage is an *approximate* indication and by itself, does not guarantee correct execution in the covered states of hardware. One would need to rely on formal proofs or compliance with golden reference models for that. Most importantly, coverage and verified execution of the benchmarks that effected the coverage only guarantee the hardware is bug-free so much as coverage thoroughly assesses all possible permutations of the inputs (instructions in the test vector) and the states of all the other hardware elements around the chosen coverpoints.

---

[1]Finite State Machine (FSM) coverage metric computes coverage on FSMs identified in the design. FSMs are usually a result of complex human coding than automated lowering from higher level hardware description languages.

2.2.1 Cosimulation and Formal Verification

Cosimulation is the practice of simulating two instances – one of a hardware under test and another, a reference model – with the same input vectors for comparing execution traces. Functional verification conducted this way is simpler and faster when there is a reliable reference model, often called a golden-reference, in the form of ISA simulators for example. The alternative would be to have mathematical or formalized model of the hardware derived through formal specifications while ensuring fidelity, or a set of formalized properties an implementation has to respect, and evaluating the model properties.

2.3 A Case for Self-Contained Hardware Verification

A well-reasoned combination of coverage metrics as critical feedback for fuzzing closes the automation loop in verification – with a few additional elements in the setup.

2.3.1 Program Generator

Firstly, we need a controllable program generator that can generate semantically, syntactically valid, executable test vectors[2]. Valid and executable test vectors are Instruction Set Architecture (ISA)-compliant and execute an appropriate set of setup procedures for the hardware under test involving, for example, register file initialization, setting up execution privilege modes, page tables, stack, and trap-handlers. This is an essential quality of generated test programs that constrain execution to well-defined and supported execution pathways. There are of course, deviations from these pathways that need to be tested depending on the hardware and the architectures in question. Design-specificity of generated programs is another quality that needs to be taken into consideration during program generation as customized features of the hardware can only be exercised by specialized program generations. The generated programs can be either in binary – in which case they can be readily executed, or in assembly or higher-level languages – in which case, they need to be

[2]Test vectors are minimal programs that can be readily executed on the hardware under test

compiled and linked into binaries for execution. A good example of a program generator is RISC-V DV [34]. The controllable aspect of a program generator is important for generating targeted test cases that can be used to target specific components or hitherto-untargeted components. The randomization ability of the generator helps span as much of the hardware state-space as possible. Targeted program generation, on the other hand, focuses on a subset of the state-space to increment coverage methodically – often driven by a fuzzer.

2.3.2 Program Loading and Execution Environment

Secondly, we need a supporting application execution environment, or a software stack capable of loading the binary, responding to environment calls, and facilitating IO operations. An example of such an offering with RISC-V is the RISC-V Proxy Kernel [5]. ZP Cosim provides a control program for loading program binaries and limited support for environment calls.

2.3.3 Mutation Engine

Thirdly, we need a mutator. A mutator is a software entity that induces a minor change in the test vector – while, ideally, maintaining the validity and executability of it – for a subsequent rereun on the hardware for an increment in progress towards a verification goal – usually, coverage. A mutator can be imagined in two different ways:

- A mutator analyzes coverage over the hardware states and, optionally, other forms of critical feedback, and turns specific knobs in the program generator (hence the controllability requirement of it) to create more targeted test cases for coverage closure.

- Another way to define a mutator is that a mutator operates directly on the generated program and effects mutations on the test vector. Seemingly innocuous random mutations such as instruction reordering, and changing the register operands and opcodes, can often reveal tricky microarchitectural bugs.

In some cases, the program generator can be tightly coupled with a mutator to compose a fuzzer. AFL++ [43] is a popular example in software testing. FuzzFactory [87] is more in the context of fuzzing for hardware verification. There is active research [95, 83, 29] in the area of effecting reliable mutations – some exploring Reinforcement Learning models to effect learnable mutations.

2.3.4 Debug Infrastructure

Finally, there needs to be support for saving *interesting*[3] test cases for fuzzing and mutation insights, and saving failing[4] test cases for later, usually offline, debug. Offline debuggability necessitates reproducibility of failing or interesting test cases.

This series of enablements allow for an iterative, self-contained verification infrastructure which forms the ultimate target of ZP Cosim.

2.3.5 Cheaper, Modifiable, Portable Verification Infrastructures

Commercial solutions to reliable processor verification are expensive and unfavourable to modifications. Open-source alternatives, on the other hand, mitigate this problem by being free and easily modifiable. However, open-source solutions usually lack reliability and technical support structures to be useful in commercial settings. Research use-cases, however, benefit hugely from the advantages of the open-source solutions. Open-source and standard interfaces to hardware-under-test, and easy debuggability of tools and community-driven bug-fixes in the tools is also greatly advantageous.

With this motivation, the thesis transitions to describing ZP Cosim.

---

[3]Interesting test vectors are those that have effected coverage over hitherto unexercised states of the hardware.

[4]Failing test cases are the intended products of fuzzing which need to be analyzed to isolate the bugs or defects from allowed implementational differences or tolerable functional behavior, or unintended software-related defects.

Chapter 3

ZP COSIM

One of the primary alignments of ZP Cosim is towards the previously described self-contained hardware verification. As such, Figure 3.1 illustrates the ambition. Notice that the blue fuzzing loop drives towards coverage convergence, and the pink verification loop kicks in upon bug-encounter for fixing the bugs discovered as a result of co-emulation. The tools exercised on the host support the identification of coverpoints, reduction of the coverpoint state-space formed, inclusion of user-provided coverpoints (not implemented in ZP Cosim), instrumentation of the original design wrapper, and FPGA synthesis and bitstream generation.

The PL is the FPGA on-board, which has a synthesized FPGA-shell enveloping the Processor-Under-Test (PUT).

The PS (the Processing Subsystem), which is a host processor for the FPGA in close proximity, receives the bitstream, flashes it on the FPGA, launch the control program which:

- Orchestrates loading of test programs on both the PUT on the Programmable Logic and an instance of the Dromajo golden-reference ISA simulator, iteratively stepping through the PUT's execution trace, cosimulating it with Dromajo, and analysis, and,

- Coverage extraction from the PUT upon program termination, processing of the coverage, derivation of mutations (which has not been implemented on ZP Cosim, and program-generation or benchmark step-through.

In case of a mismatch, the PS would save the failing test program for later offline debug through software RTL simulations for greater visibility into the microarchitecture.

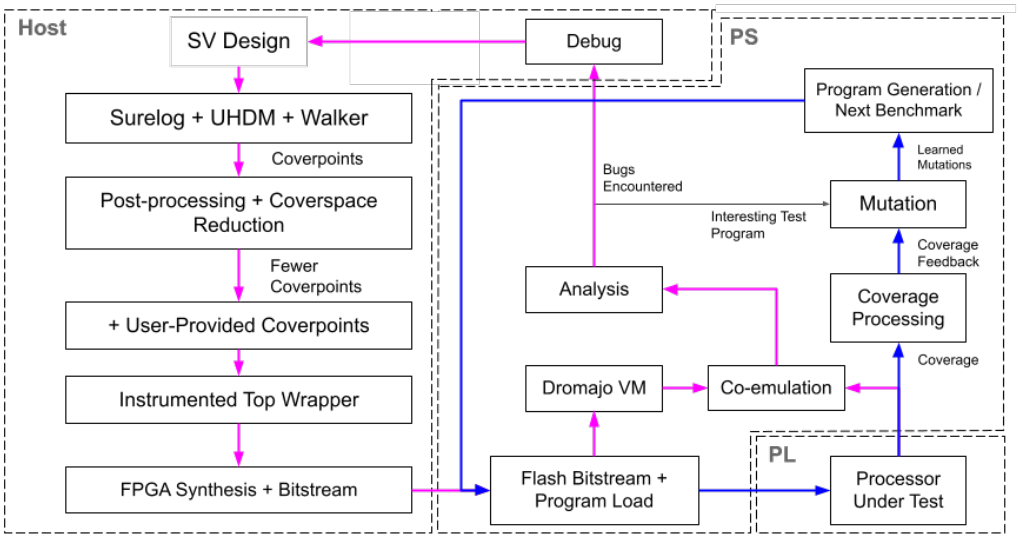


Figure 3.1: Reference Block Diagram for self-contained, iterative verification. The blue loop depicts the fuzzing loop, and the pink loop depicts the verification loop.

ZP Cosim builds on the existing Zynq-Parrot infrastructure [9] that enables rapid FPGA prototyping for BlackParrot processor on Zynq 7000 and Ultrascale FPGA devices. The following sections methodically describe the infrastructure of ZP Cosim and detail the nuances that make ZP Cosim a highly-reliable, portable, cost-effective, rapid verification research platform.

3.1 Automated Coverage Instrumentation

Coverage is an important aspect of hardware verification. ZP Cosim introduces a new automated coverage collection implementation with the help of existing open-source offerings. The fundamental requirement of coverage instrumentation is identifying key design elements whose states need to be monitored during simulation or emulation for inferring the coverage effected by test programs. Depending on the coverage metric in question, the design elements can be:

- Structural elements in the code, like expressions in assignment statements, case statements, assertions, etc., or,

- Functional elements in the design, like complex composite expressions corresponding to specific functionalities of the hardware.

The literature on which coverage metrics perform well in which situations is unclear, and there is much research insights to be had on this frontier [103].

In ZP Cosim, we demonstrate and implement automated *Mux Toggle Coverage* popularized by Kevin Laeufer et. al. [81], and a proof-of-concept case-statement coverage.

Mux Toggle Coverage is similar in spirit to the popular branch-coverage in software testing. The idea behind Mux Toggle Coverage is that muxes are the sources of multiplicity in hardware states – i.e., being the fundamental functional switches (or decision statements), the cumulative combination of toggles in mux-selects across all the muxes amount to all the different possible control paths in the hardware. The cumulative combinations are usually identified by the cross-product of the toggles on all the (interacting) muxes, and this is the truer indication of the actual coverage compared to individual toggles. Later works [63] explore other coverage metrics for efficiency, applicability, and performance reasons.

Case-statement coverage is a structural/code coverage metric that assesses the completeness of a test vector in exercising all the `case` statement control paths. It is not a standalone metric; it simply provides another perspective on one frontier of verification progress.

3.1.1 Implementation

In order to identify muxes to monitor for coverage collection, ZP Cosim packages a parameterized, modifiable walker that parses the hardware design description and outputs hierarchically referenced[1] full-pathnames of mux select-signals in the design. There are three

[1]Verilog or SystemVerilog's hierarchical referencing is a mechanism of referencing identifiers such as wires or registers outside of the scope of the module they are referenced in, by prefixing the instance names of connecting scope hierarchies. Xilinx tools support the synthesis of hierarchical referencing.

parts to the coverage metric implementation:

- Surelog [37], an open-source SystemVerilog parser and compiler that takes in a SystemVerilog hardware description of the design and generates a walkable parse tree-like object,

- Universal Hardware Data Model (UHDM) [38], the data object produced by Surelog that can be thought of as an elaborated intermediate representation of the hardware design description, and,

- A customizable walker that can operate on the UHDM object to produce the hierarchical instance names of the muxes, or coverpoints[2], for monitoring coverage.

The coverpoints generated thereof, need to be assembled into the top-level wrapper of the hardware design. This can be done in two ways:

- For simulations: SystemVerilog assertions that can be assessed through debug prints and debugged via waveform dumps, or,

- For emulations and post processing capabilities: Wired into dedicated coverage collection modules with memory modules for storage of coverage information during emulation, and read interfaces for retrieval from test environment post emulation.

ZP Cosim provides GNU Make routines to automatically package the identified hierarchically referenced coverpoints by grouping and wiring them into coverage modules in the top-level wrapper of the design in consideration.

---

[2]Coverpoints are boolean or multi-bit expressions (wires or registers) in the Hardware Description Language that are assessed, usually for toggles, for determining the coverage of a test case on the hardware in question.

3.1.2 Surelog

Surelog [37] is a SystemVerilog pre-processor, parser, elaborator, and UHDM compiler. Surelog adheres to the SystemVerilog 2017 standard [13]. Surelog also provides IEEE Design/TB C/C++ Verilog Procedural Interface (VPI) [40] and Python AST Application Programming Interfaces (APIs). In ZP Cosim, Surelog functions as a parser, elaborator, and compiler that takes as input, a file list of distributed hardware design description in SystemVerilog, among other crucial parameters, and generates a walkable UHDM object. Among the parameters taken, some of the important ones are: the specification of the top-module, SystemVerilog library extensions, compilation ordering, parameter value overrides, a switch to request full or folded UHDM elaboration, and trace and debug options.

3.1.3 Universal Hardware Data Model

UHDM [38], the product of compiling SystemVerilog hardware designs, is a representational model of the input design according to the IEEE SystemVerilog Object Model. UHDM can be operated on though the provided VPI interfaces. The provided default visitor, invokable via `uhdm-dump`, can dump out the elaborated UHDM object in textual, human-readable format.

3.1.4 Coverage Walker

ZP Cosim's Coverage Walker [85] is a parameterized visitor written to walk the compiled UHDM model of an input processor design, to identify key design elements for use as coverage indicators. The coverage metric chosen dictates the walk algorithm. The walker holds C++ STL list data structures for saving the key design elements identified during the walk which are ultimately written to an output file.

The following subsection limitedly describes the walk for the mux-toggle coverage metric as an example.

3.1.5 Implementation of Mux Toggle Coverage Metric

Recall that for the Mux Toggle Coverage metric, the walker identifies multiplexer select-signals in the design. Depending on the parameters passed into the walker, either the entire design of a specific subset can be walked, and muxes therein, identified. ZP Cosim's implementation of the walker identifies muxes by parsing the structural hardware description.

In behaviourally described designs, muxes can manifest through `if-else` statements, `case` statements, ternary statements, and occasionally, implicitly. Note that sometimes, in longer `if-else` statements, priority encoders are inferred in place of muxes; and other logic optimizations can affect inferring muxes. Despite deviating from the metric, the ultimate goal of establishing the indicators of different possible states of the hardware is still preserved.

Of the UHDM-provided VPI APIs, there are 3 main APIs utilized while walking the generated UHDM:

1. Obtaining and releasing handles to UHDM objects via `vpi_handle` and `vpi_release_handle`,

2. Iterating through multiple UHDM objects at a depth under a chosen object type via `vpi_iterate` and `vpi_scan`, and,

3. Obtaining object or design element attribute attributes via `vpi_get` and `vpi_get_str` APIs.

When using the VPI API Within the UHDM abstraction, every node of the tree is a unique structural code element that has its own VPI handle. The VPI handles are unique and exhaustively refer to all elaborated code elements. Every node also contains pointers to its children nodes – each with their own VPI handles. The root of the tree is the design instance node. The top module(s) are children of the design instance node. A typical walker starts with the handle of the design and the top-module nodes thereof, obtains VPI handles for relevant children nodes and recurses until there are no more children nodes. Relevant children nodes can be nodes that correspond to:

- Submodules, which begin a new recursion,

- Generate blocks, which elaborate and begin new recursions,

- Procedural blocks which can hold the aforementioned structural elements which need to be collected algorithmically, and,

- Continuous assignment statements which can hold ternary statements.

Figure 3.2 represents a flow diagram for identifying almost[3] all of the code structures birthing muxes in the elaborated UHDM object.

Coverpoints can be identified in two different ways in the design process:

- During design time, design engineers can manually identify and establish key coverpoints in the design for functional coverage. The obvious advantage here is the easier expressed complex functional coverpoints, and the concision of the coverspace[4] thus formed. The disadvantage is the manual nature of the exercise, and the potential of human oversight.

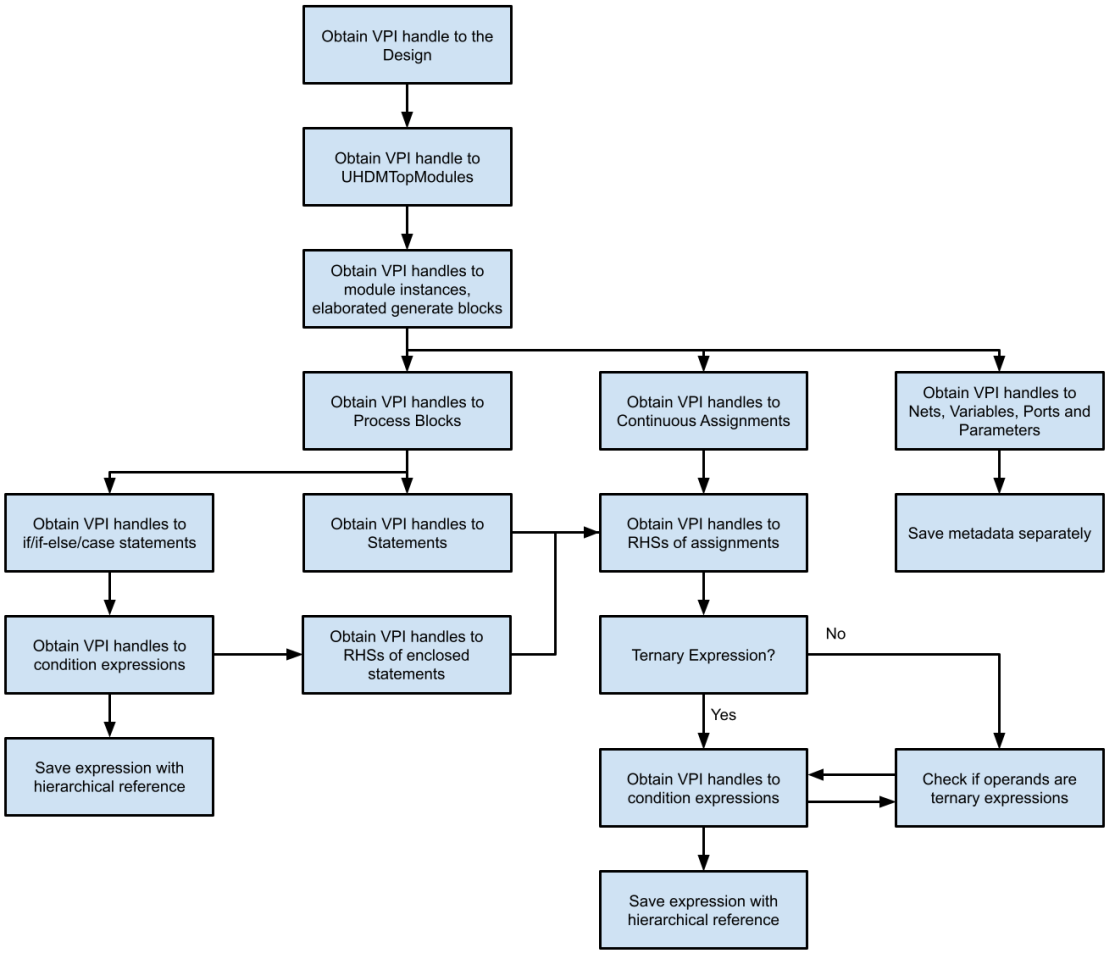
- During compile time, design compilation tools such as VCS and Verilator, can identify key coverpoints based on common coverage metrics such as line-coverage, expression-coverage, etc., with the hardware description, or more accurately, an Intermediate Representation (IR). The synthsized netlist might also offer key coverpoints for other verification objectives. The advantage is the more-complete coverspace and the potential for algorithm-based automation. The disadvantages are the redundancies, irrelevant coverpoints, missing of potentially important complex functional coverpoints in constrained coverspaces, and the resulting increased coverspace in general.

---

[3]A drawback of the implementation mentioned in Section 4.3 discusses a gap in the algorithm.

[4]Coverage state-space

Figure 3.2: Flow Diagram of ZP Cosim's Automate Coverage Metric Walker



ZP Cosim implements compile-time, automated identification of coverpoints. There is planned support for design-time source-level coverpoints via user-provided SystemVerilog `coverpoint` and `covergroup` constructs identifiable, during the walk, through their UHDM object VPI handles.

3.1.6 Coverspace Preprocessing

Coverpoints identified automatically usually suffer from duplicate or redundant, and uninteresting coverpoints.

Duplicate coverpoints : Duplicate coverpoints are a result of duplication both through structural duplications such as through multiple module instances or generate blocks, and through inherent duplication of branch conditions that are based on common wires or boolean expressions. While the latter can be blindly ignored, the former could be important depending on how the coverpoints interact external to the instances they are defined in. ZP Cosim's post processing eliminates duplicate coverpoints.

Uninteresting coverpoints : For example, values of an operand register in the ALU, could be interpreted as coverpoints in certain automated metrics – one example being control-register coverage [63]. However, they are not as important as other control-flow-related coverpoints that refer to branches and other critical functional properties of a design description. ProcessorFuzz [33], for instance, identifies such unimportant coverpoints in the FPU operands that were found to derail coverage computations in an existing implementation [63], and ignores them by construction (by only looking for Control and Status Registers and transitions). Two more examples are the misidentification of reset-conditions as mux-select signals, and the identification of parameters and constants based conditions which never toggle. ZP Cosim's post-processing also eliminates reset-conditions, and ignores parameters-only and constants-only conditions during the walk. And since ZP Cosim only identifies control-flow-related coverpoints, they are never uninteresting in the metric chosen.

```
1 wire   sel    = x | ~x;
2 assign result = ~sel ? e1 : e2;
```

Listing 3.1: Example of unreachable coverage targets

3.1.7 Coverspace Optimization

In general, coverpoints identified in the Mux Toggle Coverage metric in the aforementioned manner suffer three major drawbacks:

Drawback 1 The post-processed coverspace is still large

Drawback 2 Many of the muxes identified this way suffer from aliasing issues

Drawback 3 Covering the identified individual coverpoints singly does not imply convergence[5]. In fact, assuming there are N coverpoints identified, the coverage over the individual coverpoints would indicate exercising of $2N$ hardware states. However, the true hardware state-space is actually the cross-product (i.e., cartesian product) of the N states, equal to 2^N.

Following subsections describe how ZP Cosim fares against the above drawbacks.

3.1.8 Unreachability Analysis

Firstly, the large number of coverpoints are unreachable. Unreachable coverpoints are those coverpoint boolean expressions that cannot be satisfied within the normal operation of the hardware. For example, in the below hardware description, One coverpoint that is identified automatically in Mux Toggle Coverage metric, is ∼sel. However, it is not a satisfiable expression because ∼sel can never be true, and so, the coverpoint is never covered. When

[5]Coverage convergence is a term used to indicate a state of testing where all the possible hardware states are tested.

the root expression [6] for the coverpoint is available, unreachability can be determined by framing it as a boolean satisfiability problem. Algorithms, or tools thereof, for solving such problems are called SAT solvers, and today's SAT solvers are capable of determining the satisfiability of an expression of millions of boolean variables [86]. Sometimes design compilation often eliminate some of these cases through optimizations such as dead code elimination (DCE) or constant folding.

In ZP Cosim, Synopsys's VC Formal tools-suite with a dedicated tool, called the Formal Coverage Analyzer (FCA), helps determine unreachability of coverpoint expressions. FCA determines unreachability using proprietary constraint solvers. In addition, for the coverpoints in question, the root expressions does not have to be manually isolated – FCA leverages the VCS-compiled design representation to derive the root expressions for satisfiability checks. FCA formally proves that certain uncovered coverpoints in coverage goals (sum total of coverpoints) are indeed unreachable with the expression hierarchies that derive the coverpoints.

ZP Cosim provides a GNU Make routine to invoke VC Formal's FCA tool to automatically analyze input hardware designs, and to output an unreachable coverpoints file, called the exclusion file. This process is compute-intensive to carry out on large designs. However, having a simulation database, with randomly covered coverpoints in the design makes the process significantly faster. ZP Cosim also provides this option within the Make routine.

Once the unreachable coverage targets are determined, they can be safely excluded. This might help reduce the coverspace, thereby saving manual effort from verification engineers or fuzzers. Quantitative evaluation is presented in 4 shows that unreachability analysis does in fact help reduce the coverspace, albeit a little.

---

[6] Root expression is an in-house term for expressions in the hardware descriptions that are composed purely of constants and variables that can be externally set or determined. For example, in code listing 3.1, assuming x is an input to the module of that logic, the expanded expression of ∼`sel` is `x & x | ~x`.

3.1.9 Coverage Aliasing

Mux Toggle Coverage metric generally overlaps in the coverage information it provides. This is because many muxes are coupled to common wires and registers in the design. This introduces aliasing in coverage data that can be reduced with additional processing of coverage metrics[7]. Currently, ZP Cosim does not provide any support for detecting or reducing aliasing.

3.1.10 Toggle-Only Coverage on Coverpoints

Toggle-only coverage is the default case of assessing coverage over individual coverpoints in isolation. Because correct execution on hardware is dependant on interactions of many hardware design elements in the right states and transitioning to the right states individual assessments of design elements as implied by individual coverpoints do not cover the full scope of the hardware functionalities.

Despite seeming not very helpful at first glance, toggle-only coverage of the coverpoints identified can, at times, provide crucial insights on benchmark simulations on hardware targets – mainly, coverage holes, which are uncovered or untoggled coverpoints in the design. Chapter 4 discusses some insights obtained through analyzing toggle-only coverage effected by benchmarks on the BlackParrot processor. Some of the bugs discovered were in fact due to the insights had from this.

---

[7]Coverage Aliasing is a term to refer to instances where a coverage target is correlated with one of more other targets and can be proven to be fully covered when the correlated targets are fully covered. For example, a coverpoint `A` aliases with two coverpoints `A&B` and `B`, because when `A&B` has covered 0 and `B` transitions $0 \rightarrow 1$ or $1 \rightarrow 0$, both values of `A` are covered; therefore coverpoint `A` becomes redundant. Redundancies in the coverspace can unnecessarily increase instrumentation costs and may occasionally lead to increased verification effort and time; however, redundancies do not affect verification's reliability or functional correctness in bug finding. Nevertheless, for the former reason, it is considered crucial to reduce redundancies in coverage statespace. Aliasing in Mux Toggle Coverage can be reduced in design compilation time by assessing common ancestry of mux select signals and isolating unique and independently driven muxes.

3.1.11 Localized Cross Products

The third drawback mentioned is a bit more broader scoped. As mentioned previously, individual coverpoint assessments are not extremely useful. Taking cross-products on coverpoints is one way to expand the relevance of coverage to more realistic testing goals. Cross-products take into consideration, the states of the interacting design elements in combination, and often, those are significantly more revealing of bugs. Often, the coverage map corresponding to the cross product of the coverpoints thus obtained, are hashed down for better storage footprints on the hardware [63].

The major drawback of assessing cross-products are that they explode the state-space of coverage exponentially as described in Drawback 3. ZP Cosim overcomes this limitation by limiting the cross-product scan window. The key insight from the coverage obtained on running SPEC benchmarks is that the coverpoints do not interact exhaustively – i.e., not every coverpoint interacts with every other coverpoint. Covering non-interacting cross-products is then a redundant exercise. For example, the cross product of two coverpoints – one corresponding to a mux in the FPU, and the other to a mux in the integer multiplier functional unit – do not need to be covered because the state of one of them most definitely does not affect the state of the other in any functional sense[8]. ZP Cosim allows for grouping coverpoints and performing the cross-product within the groups. This reduces the coverage state-space (in implementation) to:

$$\frac{\lceil \frac{N}{G} \rceil \, 2^G}{2^N}$$

where N is the number of coverpoints, G is the grouping size with $G << N$ and $\lceil x \rceil$ is the ceil of x. In Chapter 4, we quantify the reduction with actual values. The obvious trade-off for the reduced coverspace is the potential of missing bugs in ignored crossings of coverpoints.

---

[8]Presumably, this exercise could still be useful in training Machine Learning models to derive and isolate mutations on the program to encourage exploration of new coverspace.

3.2 Simulation

A concise enough coverspace for useful coverage feedback enables more reliable hardware verification. Verification of a hardware target can be done through simulation or emulation. This section talks about simulation.

3.2.1 Dromajo

Dromajo [122] is a formally proven RISC-V RV64GC[9] functional reference model and a multi-CPU simulator written in C++ that is extensively verified. Dromajo allows for specification of operable memory address ranges, any IO mappings therein, bootrom sequence, and ISA extensions and custom extensions support. One of the most important features of Dromajo, that ZP Cosim considers critical is its ability to checkpoint a snapshot of the simulating system state for later resumption, thus allowing replay of key sections of long-running programs. In addition, Dromajo provides APIs for cosimulation which is crucial for ZP Cosim. Cosimulation in Draomjo is discussed in subsection 3.2.2.

Dromajo, by itself, can be useful for quick insights on ISA-level coverage effected by benchmarks. For microarchitectural coverage on the hardware design elements, which is a more complete indication of coverage, ZP Cosim leverages Dromajo cosimulation with an instance of the processor providing microarchitectural coverage information.

3.2.2 Cosimulation

Cosimulation is the process of simulating two (or more) instances of the same design with, generally, the same inputs. The expectation is that if they are both compliant with a formalized specification of the implementation, the observable states at the granularity of architecturally visible event boundaries such as instruction retirements, CSR updates, etc.,

[9]G and C are initials of extensions to the base RV64I ISA – G is a stand-in for I, M, A, F, and D. I is the base integer ISA, M is the multiplication extension, F and D are single and double precision floating point instruction extensions, A is the atomics instruction extension, and C is the compressed instructions extension.

would match within reason. At the very least, deviations would indicate one or both of the instances are not compliant with the specification.

In ZP Cosim, we explore a few different options for cosimulation. One constant instance is the Dromajo functional reference model, treated as a golden reference model for simulating BlackParrot. For the other instance, there is support for simulating BlackParrot with the ZP Cosim infrastructure in Verilator [100], Synopsys VCS [12], and emulation of ZP Cosim on the FPGA which is discussed in Section 3.3.

Verilator Verilator is an open-source Verilog and SystemVerilog simulator that works by converting the input hardware designs to functional, optionally multi-threaded, C++ functional models (among other options) that can be compiled together with an instantiation code (generally in C++) and other optional C++ functional models for IO interactions. Execution of the compiled binary simulates the hardware functionalities. In ZP Cosim's case, Verilator generates the functional models for the ZP Cosim hardware infrastructure (synthesizable on the FPGA), and BlackParrot. The compilation also takes in the control program, Dramajo's Cosimulation libraries that instantiate the Dramajo virtual machine and provide APIs for stepping through program execution and comparing the executions. More details on the control program is available in Section 3.3.

Synopsys VCS Synopsys VCS is a high-performance commercial functional verification solution widely used in the industry. ZP Cosim utilizes VCS simulation support for simulating hardware designs. Like Verilator, VCS also allows for linking C/C++ models with the hardware design description.

The inputs to the cosimulation instances are managed by the control program, which itself takes the executable program binary as the input along with other parameters. For comparison of execution, ZP Cosim relies on the default trace comparison option provided by Dromajo. The essence of trace comparison is that the execution trace, consisting of the

retired instruction word, the side-effects on the hardware (registers in the Register Files written, important CSRs modified), the program counter. Any asynchronous events encountered during execution such as external interrupts, and unmodelled IO interactions, if any, are manually overridden in Dromajo to match the state of the instance of ZP Cosim being simulated. More details on this is available in Section 3.4. In addition, the trace comparison is done in "almost"[10] real time so as to reduce the amount of time spent on early-fail test cases, and lessen the deviation of hardware state from the point of bug, which improves human debuggability and the deviation of the coverage.

3.3 Emulation: Zynq-Parrot

Zynq-Parrot [9] was conceived as a customizable FPGA shell to present a flexible AXI wrapper around accelerators and soft processors such as the open-source BlackParrot processor [91], to facilitate easy interfacing with the host Zynq-7000 series FPGA boards. At its current state, Zynq-Parrot has evolved into a rapid emulation/prototyping solution for more processor designs on more FPGA boards, along with a host of new features and capabilities. In this thesis, however, we restrict our discussion to evaluating BlackParrot on the Avnet Ultra96v2 [10] FPGA board featuring Zynq Ultrascale+ Multiprocessor System-on-Chip (MPSoC).

3.3.1 Infrastructure

Zynq-Parrot's hardware system involves the Xilinx Zynq Ultrascale+ MPSoC component with a customizable FPGA shell. In Xilinx's nomenclature, the MPSoC component is referred to as the Processing System (PS), while the FPGA component is referred to as the Programmable Logic (PL). In the Ultra96v2 board, the MPSoC is the Xilinx Zynq UltraScale+ MPSoC ZU3EG A484 which comprises of a quad-core ARM Cortex-A53 processor with support for ARM's Single Instruction Multiple Data (SIMD) extension, called NEON. The MPSoC also comes standard with 2 GB (512M x 32) of LPDDR4 memory and Delkin

[10]It is not exactly real-time because of the FIFO interfaces (asynchronous and synchronous) used to communicate the commit information.

Core Configuration	Quad-core ARM Cortex-A53
CPU Frequency	Up to 1.5GHz
Architecture	Armv8-A Architecture in A64 or A32/T32
Features	NEON Advanced SIMD instructions Single and double precision Floating Point instructions
Cost	$299

Table 3.1: Specification of Zynq Ultrascale+ MPSoC in Ultra96v2 board

System Logic Cells	154,350
CLB Flip-Flops	141,120
CLB LUTs	70,560
Distributed RAM	1.8 Mb
Block RAM	216 blocks; 7.16 Mb
	36 Kb block granularity

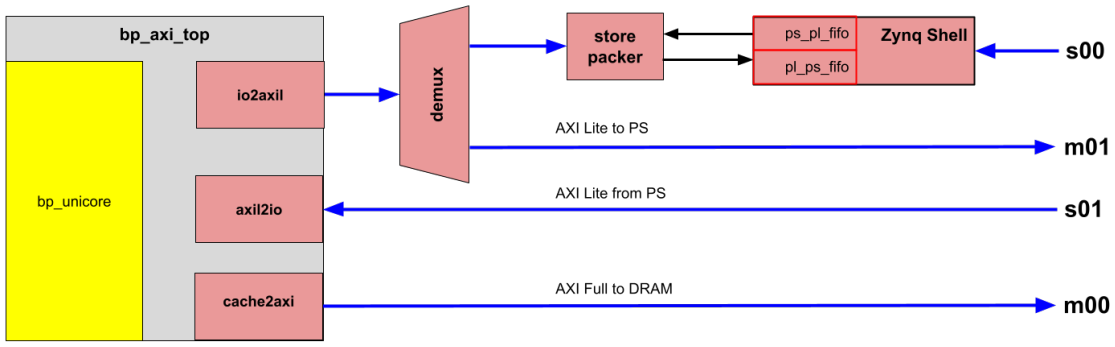
Table 3.2: Specification of Zynq Ultrascale+ PL (FPGA) in Ultra96v2 board

16 GB microSD card for storage. More details on the MPSoC is presented in Table 3.1. The details of the FPGA package is presented in Table 3.2.

The PS is capable of booting Linux and communicating with the PL through a variety of PS-PL interfaces. Zynq-Parrot utilizes 2 of 6 high-performance slave AXI interfaces in 32-bit configuration for DRAM communication (m00 in Figure 3.3), and the 2 high-performance master AXI interfaces in 32-bit configuration for communication with the FPGA shell (s01 in Figure 3.3) and the slave port of BlackParrot (s00 in Figure 3.3).

The PS runs a control program (in C++ or Python), `ps.cpp`, that memory-maps the address space of the AXI interface to user memory, and is capable of:

Figure 3.3: Concise Block Diagram of Zynq-Parrot FPGA shell.



- Flashing the system bitstream on the PL using APIs made available by the PYNQ development environment,

- Resetting BlackParrot hardware, initializing DRAM pointers and other configuration registers in the shell (PL) by writing to CSRs in the shell.

- Loading RISC-V compliant program binaries and writing configuration registers in the address space of BlackParrot,

- Processing and responding to some environment calls, and interpreting termination signals from BlackParrot, and,

- Reading and writing FIFOs and registers in the FPGA shell for any auxiliary information like profiling data.

On the PL is the synthesized FPGA shell that wraps over a unicore configuration of BlackParrot. The FPGA shell hosts a parameterized number of synchronous FIFOs and general-purpose registers that can be read and written-to by the PS via the s01 interface.

Figure 3.3 is a high-level block diagram of Zynq-Parrot FPGA shell with the interfaces highlighted.

3.3.2 Additional Capabilities of Zynq-Parrot

Prototyping Ariane

Zynq-Parrot supports simulating Ariane[134], another open-source RISC-V RV64GC implementation. The modifications to the original infrastructure to enable support for Ariane are minimal, and the modifications are made available on GitHub [9]. This also indicates the portability of Zynq-Parrot to other microarchitectures.

Zynq Farm

Zynq-Farm[49] is a hosted FPGA cluster maintained by the Bespoke Silicon Group. The farm consists of 20 Ultra96v2 FPGAs with statically allocated IP addresses. The value of this farm is in the shared Network File System (NFS) over ethernet LAN that provides much greater shared mounted storage on the host server than on the individual FPGAs. This enables a unified view of the workspace for running multiple test programs and co-dependant fuzzing instances in parallel, reliably running extremely long programs such as SPEC benchmarks for similarly long and storage-intensive trace-based performance analysis and coverage insights, and, faster storage than the de facto offering from the SD card. The defacto SD card based storage caps access speeds to around 95 MB/s read and 55 MB/s write [21], whereas with the farm, the storage speeds would only be limited by the network bandwidth and the host's SSD speeds.

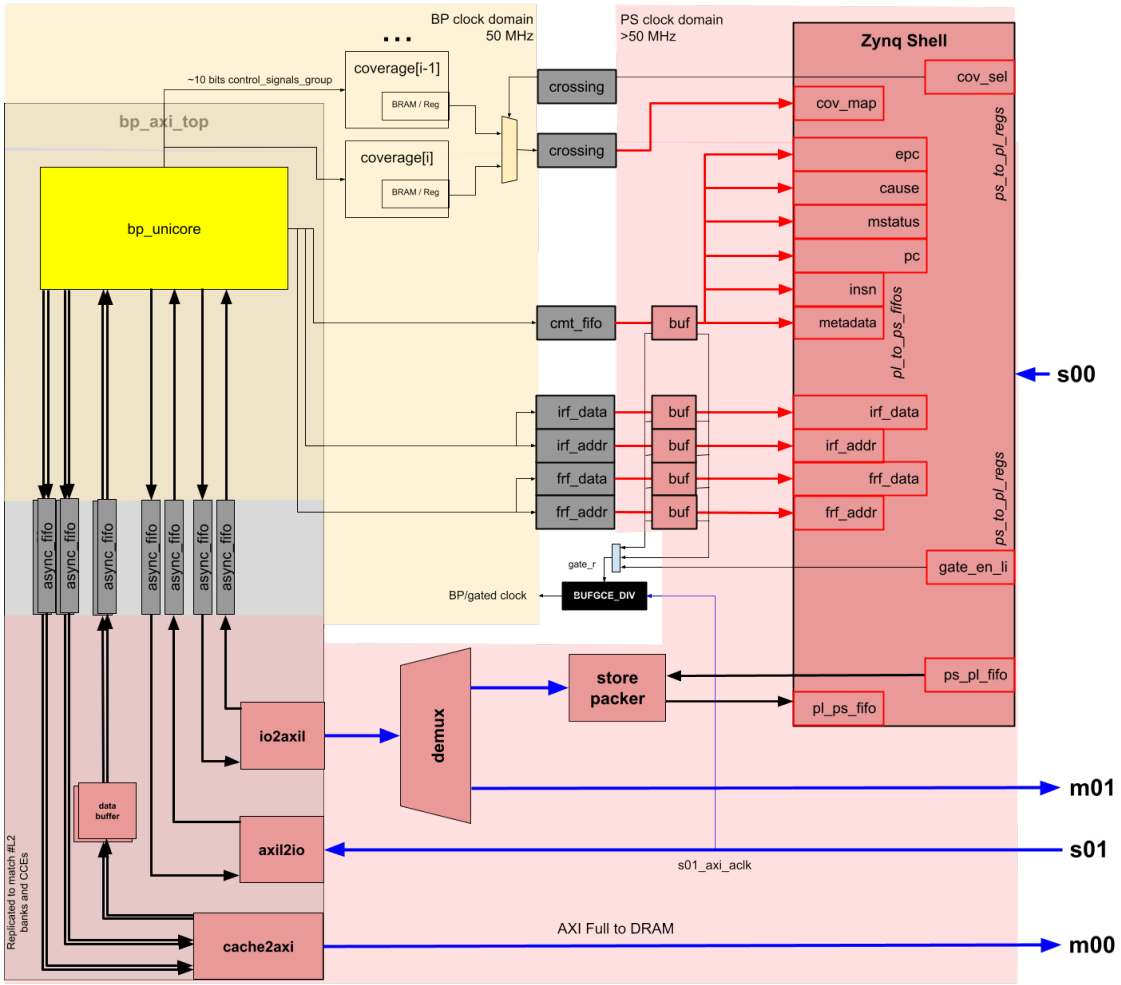
3.4 ZP Cosim

Building on the existing Zynq-Parrot infrastructure, ZP Cosim provides enhances support for FPGA-cosimulation, coverage extraction, and enablements for additional features.

3.4.1 Description of the Modified FPGA Shell

Figure 3.4 is a concise representational block diagram of the ZP Cosim FPGA shell. ZP

Figure 3.4: Block Diagram of ZP Cosim's modified FPGA shell.



Cosim modifies the original Zynq-Parrot shell in four important ways. The reasoning and the enablements thereof are also discussed inline:

1. **Execution Trace Transfer for Cosimulation** In order to transfer the execution trace from the FPGA to the control program on the PS running an instance of Dromajo, ZP Cosim introduces additional data paths for buffering and transferring execution trace information. ZP Cosim allows for the transfer of the following required variables for cosimulation (in addition to the existing resources in the shell):

    - Program Counter (`pc`) is used for comparing the address of the retired instruction.
    - Instruction Word (`insn`) is used for comparing the actual instruction executed. Along with the PC, it ensures that the right instructions are fetched from the right addresses in the (instruction) memory.
    - Commit Data (`irf_data, irf_addr, frf_data and frf_addr`) are used to compare the effects of the execution on the hardware. Instructions that commit data to either the Integer Register File need to match the register written (`irf_addr`) and the written data (`irf_data`) with those of Dramajo's. Similarly the Floating Point Unit, in BlackParrot, has its own FPU Register File, the changes to which need to match with those of Dromajo's. This is important for both correctness of execution in instruction granularity and for localizing bugs to specific instruction executions.
    - Machine Status (`mstat`) is used to compare the machine hart's[11] current operating state which includes status information on interrupts, functional units, execution privilege modes, among other things. Matching machine status is important in debugging as execution can differ significantly when there are changes in the machine status which needs to be debugged separately.

[11] A Hart is a RISC-V terminology to refer to a hardware thread, or equivalently, a hardware execution context that holds its own independantly operable set of states mandated by the RISC-V ISA. It differs from a core in that a core can host one or more harts.

- Interrupt/Exception Data (`cause, epc`) and Instructions Retirement Count (`minstret`) includes the cause of the exception or interrupt (which also indicates if it's an exception or interrupt) which is necessary for adjusting Dromajo execution to replicate the events, and the exception PC so Dromajo may take the exception when appropriate. Note that the instruction retirement count also needs to match for Dromajo to take the exception at not just the right PC, but also the right repetition of that PC. This is important because the FPGA shell greedily enqueues commit-data as and when available; it does not "stitch" the commit to the RF with the corresponding instruction retirement. For example, if a load instruction succeeds and gets retired, the shell fetches the instruction word, the PC of the retired instruction and the mstatus, and enqueues them into the commit FIFOs. Only later, when the load is serviced, an update to the register file gets enqueued into the commit FIFOs. This allows for better decoupling of the two pieces of the commit information.

Porting ZP Cosim to other RISC-V implementations such as Ariane [134] requires identification and wiring of the aforementioned data structures from the implementation to the shell infrastructure. With SystemVerilog, there is support for hierarchically scoped variable wiring which obviates needing to modify the original RTL. Note that each of these are specified as `pl_to_ps_fifos` – parameterizable number of FIFOs provided by the shell. In Figure 3.4, you can observe each of the above FIFOs within the Zynq Shell on the right. The dequeue of the FIFOs happens through the slave AXI Lite interface, `s00_axil`.

2. **Asynchronous Clock Domains** This modification separates the clock to the FPGA shell from the clock to BlackParrot, and enables asynchronously[12] clocked DDR and BlackParrot's IO interfaces. Because ZP Cosim compares the execution trace in real

[12]Asynchronously to the processor's clock

time during emulation, there is continuous exchange of data between the PL and the Draomajo virtual machine instance running on the PS. The rate at which the shell communicates with the PS (determined by the AXI clock to the shell) and the speed of execution of the processor under test, BlackParrot (determined mostly by the clock to BlackParrot), need to be decoupled so that each may run at their maximum potential independently. In Figure 3.4, the two clock domains are represented by the red and yellow shaded polygons. Understandably, this introduces two problems:

- Clock domain crossing[13], which needs to be specially handled. ZP Cosim uses dedicated asynchronous FIFOs from the BaseJump STL [119] – `bsg_async_fifo`. It implements gray-code pointers to synchronize FIFO pointers across clock domains for correct valid and ready signal generation [36].
- Because of the different rates of enqueuing and dequeuing across the clock domains, there is a potential for data drop when the rate of trace generation is greater than the rate at which the asynchronous FIFOs are dequeued. The inverse case is handled via a FIFO (occupancy) counter presented as a register to the PS – essentially, when the occupancy of the FIFO is non-zero, it would be safe to dequeue the FIFO, and so if the rate at which the control program dequeues the commit FIFOs is higher, the control program loops back around on a waste cycle.

From Figure 3.4, notice that all of the trace data crossing the source clock domain are crossed safely through BaseJump STL's asynchronous FIFOs, hereon referred to as the commit FIFOs for simplicity (dark gray boxes in the junction between the two clock domains). Each of these FIFOs are also 32 elements deep to enable contiguous bursts of execution (between FIFO empty and FIFO full) which are easier to debug on a waveform viewer.

[13]The mechanism of safely crossing data from one clock-domain to another, usually by techniques such as double-latching, or via use of carefully designed asynchronous FIFOs.

Notice also that the clock domains intercede the DDR and IO interfaces with BlackParrot. These are also safely crossed with bsg_async_fifos. A few important notes here:

- BlackParrot's L2 cache is parameterizable and banked – in the configuration used in ZP Cosim, dual banked. The AXI adapter is parameterizable too, and replicates the cache-facing IO to match the number of banks so each bank has its dedicated IO to the adapter. Both the replicated interfaces are similarly safely crossed. Zynq UltraScale+ MPSoC devices can potentially have AXI Timeout Blocks with which AXI masters, when the corresponding slaves do not send responses within a stipulated time, can lock up. Even otherwise, when the slave device does not respond in time, the master would stall. It is therefore crucial to ensure a slave in a transaction responds fully irrespective of whether a gate is asserted at the source.

  The requests going out of BlackParrot's L2 cache (within bp_axi_top in Figure 3.4) in BlackParrot's clock domain into the AXI adapter (cache2axi in Figure 3.4), are 64-bits-wide packets with data generated from L2 writebacks, if any, sent separately. Read requests are sent out in a single packet, and write requests are sent out in a single write packet with as many packets of write data as the burst length supported by the interface. For write requests, there is a store-and-forward FIFO (AXI-burst-length deep) that stores data packets and forwards them when full. The incoming responses can be burst transactions (with a programmable maximum burst size; in the configuration used, it is 8 words), and so, ZP Cosim implements a dedicated buffer per bank which can hold conservatively 4 times (parameterizable) the maximum size of a transaction (or 64 words), and since there can only be a maximum of 1 outstanding request per bank, this is well within capacity. So, if BlackParrot gates mid-transaction, the implementation ensures no violations.

- Both AXI Lite interfaces (incoming and outgoing) are also safely asynchronously crossed. The AXI bus is never gated in the implementation; only BlackParrot. There are no outstanding requests possible in the AXI Lite IOs as configured, and the AXI Lite transactions are necessarily single-beat, and so there is no possibility of gating mid-transaction.

3. **Clock Gating** Clock-gating BlackParrot facilitates control of the rate of trace generation as an indirect way of handling the back-pressure from commit FIFOs. The ARM PS in the Ultra96v2 board, running Linux, is capable of dequeuing the shell at a maximum rate of 13.36 MTPS[14] for writes and 3.57 MTPS for reads. And BlackParrot, as synthesized, can operate at a (tested) maximum of 50 MHz, i.e., assuming the limiting Instructions Per Cycle (IPC) of 1.0 for the single issue BlackParrot, the maximum rate of trace generation is 50 MIPS[15]. In order to automatically match the speed of the producer (trace generation from BlackParrot) and the consumer (AXI Lite reads from the PS intp the PL) across the asynchronous FIFO, ZP Cosim opts for controlling the producer. In order to slow down trace generation, two solutions were explored:

   - Architecturally freezing the core so that there are no new instructions fetched while the FIFOs are full. This requires intrusive changes in the description of the design (reducing portability of the verification infrastructure) and can lead to complications in handling the *full* conditions of the commit FIFOs because there is the possibility of in-transit commits (mid-pipeline) getting dropped when the commit FIFOs become full. There are solutions such as early-full signalling[16] and bypass-buffering that was implemented. However, the conservatively-sized buffer is not utilized consistently and the alternative solution discussed below is much

[14]Million (32-bit word) Transfers Per Second

[15]Million Instructions Per Second

[16]Early-Full is the concept of declaring a FIFO as full prematurely so that any "limited" soon-to-be-enqueued data can still be safely accepted.

simpler.

- Clock-gating the core so that the entire activity (including in-transit commits) are held constant until the FIFO recovers space. This is the better option because it does not intrude into the RTL of the processor (BlackParrot in this case) and allows for a simpler FPGA-implementation on Ultra96 through the Xilinx-provided clock gating macros, while also saving area of the extra buffers.

The gating assertion signal (`gate_r`) is derived from the wired-OR full-signals of the commit FIFOs. However, the deassertion is derived from the wired-AND of the empty-signals of the commit FIFOs. This means any of the commit FIFOs becoming full would assert the gate, but only all the commit FIFOs becoming empty deasserts gate. This along with the previously mentioned judicious asynchronous commit FIFO buffers together allow for debug-friendly, burst-trace generation. Note that the gate assertion and deassertion signal is synchronous with the original clock, as it should be. In Figure 3.4, this is seen as the `gate_r` signal being entirely in the input (pink) clock domain. Note that in order to guarantee correct clock domain crossing, all the crossings are either synchronized through BaseJump STL's asynchronous FIFOs, or through two-clock synchronizers[17].

4. **Coverage Collection and Extraction** ZP Cosim implements the synthesizable Mux Toggle Coverage metric with supportive interfaces for coverage retrieval from the PL. The coverpoints are synthesized on the FPGA in groups. Each group is an instance of a dedicated coverage collection module each of which:

   - Take the parameters corresponding to the group size, the type of coverage (toggle vs cross-coverage), and a group ID (useful for addressable coverage retrieval),

[17]Two-clock or three-clock synchronizing is the technique of crossing usually single-bit data across two cross domains with the use of cascaded flip-flops clocked by the output clock domain. Any metastability is local to the first of the flip-flops outputs and the subsequent ones correct for it.

- Take the inputs of the individual coverpoints which are wires, a reset and a master-reset signal, and related control signals for coverage retrieval, and,
- Emit the addressable coverage maps after program execution of interest.

The inputs are registered and backwards-retimable, and the coverage collection module is flattenable, meaning the input register placement can be external to the module and can be automatically retimed to aid timing.

When toggle-only coverage is implemented, the coverage collection module houses just a register as wide as the number of coverpoints collected that OR-updates the cumulative coverage. The coverage collection is an action of reading the register. When cross-coverage is implemented, the coverage collection module synthesizes BRAM module for distributing and storing the coverage map[18].

Distributing coverage map into groups as described in Section 3.1.11 is done to also reduce the size of memories . Within each distribution, the cross-coverage state-space is rearranged to fit within standard BRAM instances. Ultra96v2 provides 36 KB BRAMs that can be used in a few different BRAM configurations. The chosen grouping of G=10 optimizes the fragmentation of the BRAM memories.

3.4.2 Control Program

Understandably, the control program in ZP Cosim has a broader functionality. There are two major modifications to the control program:

1. **Cosimulation Support** The control program initializes an instance of the Dromajo virtual machine, and supplies the same program executable that will be loaded on the PUT, along with basic configuration of the ISA. The PL presents the previously mentioned trace information through distinctly addressable PL to PS FIFOs. There

[18]Coverage map is a one-hot encoded representation of the coverage statespace represented by the coverpoints in question.

are dedicated read-only count registers that can be used to determine when to safely dequeue the FIFOs. The control program reads the FIFOs accordingly, sequentially, in a loop, into software-managed queues. This is a polling sequence as opposed to an interrupt-based sequence. When an instruction commit with all corresponding changes effected[19] are recorded, the Dromajo virtual machine is stepped one instruction, and the corresponding execution states compared. The failure of comparison halts the cosimulation and dumps out limited runtime data structures for a minimal debug. In simulation, debug exercise is significantly faster with simulation waveforms for greater visibility into the causes of the mismatches; not so much in emulation which motivates the needs to save failing tests in emulation for a later offline simulation for debug. Upon completion, indicated by a write to a certain address, the benchmark is declared to either PASS or FAIL. A FAIL would indicate correct functioning of the hardware, in compliance with the ISA, but an incorrect execution of the program.

2. **Coverage Extraction** ZP Cosim readily supports once-per-program coverage extraction, and with some modifications, can support dynamic coverage extraction through key epochs in the test-programs. Because the coverage maps are distributed and addressable, the control program can be modified to implement customized coverage extraction which supports targeted fuzzing. ZP Cosim supports sequential reading of all the covergroups. For toggle-only coverage, any non-zero write to it triggers toggle coverage dump; for cross-coverage, a valid address of the coverage map module needs to be provided, and the interface dumps the entire coverage map of the module addressed. Note that, with minimal post-processing, the toggle-only coverage can be computed from cross-coverage. The coverage extracted from the control-program is saved in files according to the benchmarks or test-programs the coverage maps were extracted from. Another Python script analyzes the coverage map and re-associates

[19]Whether an instruction commits to one of the register files is known by just the instruction word composition. In cosimulation, such effects are to be completely carried out before comparison so we compare the most up to date effects of execution.

coverpoints with corresponding hits. Ideally, at this point, the coverage information should be utilized to derive specific (valid) mutations to perform on the generated program or the benchmark to target coverage increments. This activity has been pushed to future work.

3.4.3 Solutions and Optimizations

Memory for coverage map ZP Cosim saves coverage maps in Block RAMs. The coverage modules always update a single bit (corresponding to the coverpoints hit-pattern) in a BRAM entry. In the FPGA implementation, a 1RW RAM with support for bit-maskable write is necessary. However, not all FPGAs support bit-maskable writes[20]; the FPGA on Ultra96v2 board does not support bit-maskable 1RW RAMs[21]. This is implemented with a 1R1W RAM which internally, at every clock cycle, reads the memory, sets a bit according to a mask, and writes back the update in the next cycle. This optimization is also made available in the BaseJump STL [119].

Utilization optimization for coverage map The basic unit of BRAM provides a 36 Kb storage capacity. Obliviously managing coverage maps could lead to external fragmentation, thereby leading to inefficient utilization of the BRAM resources. ZP Cosim optimizes the coverage map storage with carefully sized RAM instances, and for that, the crossing groups are appropriately chosen. For example, with a crossing group of 15 bits, implying 15 coverpoints in a grouping, the coverage map is 2^{15} bits wide. This is implemented with a 512-deep 64 bit wide memories[22].

Faster PL-PS communication Another optimization is employing ARM NEON SIMD loads in the control program to load 4 32-bit words from the PL at a time. The PS in

[20]Some FPGAs, like the one on Ultra96v2 board, only support byte masks.

[21]1RW is a RAM port configuration that stands for **1** port **R**eadable and **W**ritable. 1R1W is another commonly used configuration that stands for **1** port **R**eadable and **1** (more) port **W**ritable.

[22]In actuality, this will be implemented as 512 x 72b BRAM; unless there are further optimizations that can be employed to utilize the wastage.

the Ultra96 board supports ARM NEON SIMD operations upto 128 bits (or 4 words). This allows for the control program running on the PS to load upto 4 words of memory mapped PL to PS FIFOs and registers. However, because the FPGA shell employs a memory-mapped 32-bit AXI Lite bus to link the FPGA shell FIFOs and registers to the PS, the realistic throughput improvements are limited until the point of serialization of the SIMD loads – at the AXI Smartconnects between the PS and the PL.

Toggle-only coverage Because the cross-product on the coverpoints explodes the coverage map to be stored, and the area consumption thereof, opting for toggle-only coverage could be a tradeoff on area for being able to fit larger designs. The alternatives would be to either leverage some sort of multi-FPGA systems support [57] to be able to distribute larger designs, or have multiple FPGAs execute the same test-program and collect coverage over different covergroups and later combine the coverages.

In addition to the aforementioned optimizations, there are the following two potential optimization in the works:

Buffered Writes for Coverage Map Memories A potential optimization is to employ write-buffers to reduce total writes into the BRAMs. This can reduce energy consumption, and enable the use of 1RW memories yet again, but may not lead to improvements in utilization on FPGA. An insight obtained from Verilator simulations is that not all coverage maps are freshly updated at each cycle. Sometimes, there could even be same RAM addresses consecutively written as some coverpoints are consistently hit, and some not so much. In Ultrascale+ FPGAs, 1R1W RAMs and 1RW RAMs do not change the BRAM utilization much, but do improve energy and timing. This is because the BRAMs in the FPGAs are inherently dual-ported, and can be configured to run as single-ported by disabling one of the ports. Within dual-ported operation, there are two more subcategories – Simple Dual Port (SDP) where both the ports have a fixed read or write functionality, and True Dual Port (TDP) where either port can perform

either read or write. As designed, the BRAMs for coverage modules employ SDP memories which are already optimal. Moving to single-ported RAM thereon would lead to better energy efficiencies. However, using 1RW RAMs for bit-maskable writes leads to a problem: at each clock cycle, assuming there are incoming write requests at every clock-cycle (the worst case scenario), the RAM needs to perform a write of an updated previous cycle's read, and a read for the current cycle's write request. However, relying on the aforementioned quality of coverage maps, the existence of write-buffers could reduce the number of writes enough so that a 1RW RAM could be utilized in place of the 1R1W RAM for recording coverage maps.

However, there is still a possibility of a hazard when the write-buffers are full; potential solutions include:

- Conservative writes: When the write-buffers are full, incoming writes can be dropped. This does not affect the reliability or the functional correctness of verification, but "underrepresents" coverage – thus increasing verification time and effort. This would be just fine during the initial, coverage-explosion phase of program runs since the hard-to-hit coverpoints are less likely to be hit during this phase; they get targeted later towards the coverage saturation phases. In order to be more confident that none of the hard-to-hit coverpoints were dropped, we could re-run a few initial programs with a randomized dropping of writes so some more of the previously dropped writes could be incorporated.

- Deeper write-buffers for greater buffering capacity. Presumably, larger write-buffers would lead to larger or more comparators, which worsen overall utilization.

- Clock-gating BlackParrot when any of the write-buffers are full. This could be an overkill considering the first bullet point.

3.5 RISC-V Design Verification

Finally, ZP Cosim comes packaged with RISC-V Design Verification (`riscv-dv`)[34], a configurable RISC-V random program generator. RISC-V DV is the penultimate step towards closing the hardware verification loop – the coverage feedback obtained through emulation of a generated program remains to be connected to a mutator to dynamically reconfigure the program generator. At the time of writing the thesis, this connection has not been implemented.

Chapter 4

EVALUATION OF ZP Cosim

In this chapter, we evaluate ZP Cosim on the BlackParrot processor, and compare existing commercial and open-source solutions to full system verification.

4.1 BlackParrot

BlackParrot [91] is an open-source, silicon-validated[1], RISC-V RV64GC-compliant processor that is capable of booting Linux. Due to BlackParrot's rich feature-set [35, 94, 84, 70, 131, 33] and widespread adoption for both research and utility, BlackParrot is seeing continued academic development, which makes BlackParrot a valuable candidate for ZP Cosim's evaluation. BlackParrot is FPGA-validated and runs at a maximum clock frequency of (at least) 50 MHz on the ZU3EG A484 FPGA on the Ultra96v2 board.

4.2 Performance Evaluation

The speed of verification is an important consideration while evaluating a verification infrastructure. The pith of ZP Cosim's verification is cosimualtion, and so the speed of cosimulation is an important metric for evaluation. BlackParrot supports cosimulation against Dromajo with Verilator [100] functional simulation, Synopsys VCS [12] simulation, and, with ZP Cosim's FPGA-cosimulation.

Table 4.1, details the average baseline Dromajo cosimulation speeds with the above three simulators. Baseline corresponds to the system without any of the optimizations discussed in Chapter 3.

---

[1]BlackParrot has been taped out in Global Foundries 12 nm.

Simulator	Cosimulation Speed (MIPS)	Absolute Speedup	Emulation Speedup
Verilator	0.000177	1.00x	
VCS	0.001477	8.34x	
Emulation (with tracing)	0.011	62.14x	1x
Emulation (no tracing)	0.192	1084x	17x
Emulation (with SIMD reads)	0.273	1542x	24.8x
Emulation (with reduced reads)	0.308	1740x	28x
Emulation (with async[1] operation)	0.372	2101x	33.8x

Table 4.1: Average baseline Dromajo cosimulation speeds in various backends. The speedup factors are in comparison to Verilator cosimulation – for being the lowest.

[1] With FPGA shell/AXI bus at 80 MHz, and BlackParrot at 40 MHz. This is achieved by employing clock divider macros provided by Xilinx. It may be possible to run the bus at higher frequencies, and this only serves as a proof-of-concept of the asynchronous operation optimization discussed in Section 3.4.1

Dromajo Simulation	Simulation Speed (MIPS)
On x86_64 (3.2 GHz)	16 — 0.9 with trace on
On ARM AArch64 (0.3 GHz)	2.4 — 0.07 with trace on

Table 4.2: Baseline Dromajo functional simulation speeds on AMD Ryzen 5800H and the Ultra96v2 board. In both cases, simulation is on a single-core for the same benchmark.

Observing the time spent by the control program on various activities shows that that the majority of the time spent on an instruction cosimulation is in the 14 PS-PL memory accesses[2].

Inference: Reducing the PL-PS access time would significantly speed-up cosimulation.

Applying the optimization discussed in Section 3.4.3 – SIMD loads from the commit FIFOs, we observe that the cosimulation speed has increased by nearly 40%. Additionally, every time we read the FIFO count registers, we can save the value for the subsequent iteration where we can skip verifying that the count is non-zero which reduces the number of loads. This increases cosimulation speed by an additional 10%. The number of reads into the PL-PS FIFOs can be reduced further by 4 by combining the 5 FIFO count registers into 1 and operating with a local copy for the iteration.

For reference, the average speed of functional simulation in Dromajo is provided in Table 4.2. Functional simulation in Dromajo is useful as a reference for comparing executions of programs on architectural simulators; it does not provide any information on the correctness of a hardware implementation.

Note that, in Table 4.1, there are cosimulation speeds stated with and without tracing enabled. Tracing is for enabling debug prints and execution trace dumps – which would add

[2]Note that the 14 PL reads are: 5 FIFO count registers – for emulation status/standard library calls, the instruction commit FIFOs, the Integer RF and FPU RF commit FIFOs and the trap/exception FIFOs, and 8 data registers on average: 2 each for PC, mstatus, commit-data to the RF and 1 each for instruction word, and the register ID in the RF (either Integer RF or FPU RF; not both); this excludes the trap/exception and instruction retire counts FIFOs as these are relatively less common events.

latencies corresponding to environment calls and file IO in case of FPGA emulation. Since the intention is for the FPGA emulation to merely expose failing test programs and the actual debug to be performed in simulation offline, the requirement of tracing on the FPGA is obviated to an extent.

Also note that RTL simulation through VCS or verilator is much slower than Dromajo functional simulation, which is an ISA simulator that is implemented in software. The ISA simulator does not model any of the implemented microarchitecture and so does not provide any visibility into the actual working of the processor; it is instead used for software testing and debugging. But as a reference, it allows for sanity checking of the generated test-programs which is an important part of cosimulation – for isolating bugs within the context of valid and well-defined behaviors.

Additionally, Synopsys VCS simulator is commercially-licensed, Verilator and ZP Cosim are open-source. The speed differences (as evaluated on Intel(R) Xeon(R) Gold 6254 CPU at 3.1 GHz) between the simulators are not negligible, however, there are nuances to be appreciated. For example, there are differences in the way VCS simulates target – with ASIC synthesis as a primary goal, which leads to stricter rules on hardware descriptions and clocking infrastructure. Verilator is a fast functional simulator that is forgiving of clocking and other more synthesis related rules.

Emulation on the FPGA is clearly significantly faster than simulations through VCS or Verilator. This is the intended bracket ZP Cosim competes in. With the automated coverage extraction implemented, there is little to no change in performance. But the logic utilization on the FPGA increases significantly. Table 4.3 details the baseline utilization with group-wise crossed coverage implementation for a group size, G=10, and toggle-only coverage implementation.

4.2.1 Efficient crossing of coverpoints

From the logic utilization, we see that the FPGA shell infrastructure implementing toggle-only coverage leads to a minor increase in utilization. Note that because of the use of

Logic Block	Total Available	Baseline	Toggle Coverage	Cross-Coverage	
				Ternary St. Only	Combined
LUT	70560	56.98%	60.15%	65.03%	136.82%
LUTRAM	28800	12.77%	17.43%	31%	111.7%
FF	141120	13.90%	14.47%	18.33%	139.89%
BRAM	216	49.31%	49.31%	81.01%	106.3%
BUFG	196	1.53%	4.59%	-	-

Table 4.3: Logic utilization (in %) on the FPGA. The total number of logic blocks of each type is also provided for reference.

clock-gating, there are additional clock buffers utilized, but the utilization increase is still not concerning. Cross-coverage, on the other hand, explodes the utilization as hypothesized earlier. The full-scale cross-coverage – of 219 groups of 10 coverpoints each, does not fit within the FPGA on Ultra96v2, and so, Table 4.3 records, separately the increase for limited ternary statement-based covergroups, and all covergroups. We see that for a grouping size of 10 coverpoints in a covergroup, leading to instantiation of a 32x32 BRAM per coverage module, the increases in BRAM and LUTRAM utilization for the combined covergroups exhausts the FPGA device. Two optimizations that may be valuable are discussed in 3.4.3. A naive way to solve this problem is by having multiple FPGAs collect coverages from different covergroup sets. For BlackParrot, that would be 3 FPGA's for a combined collection of all the cross-coverages.

A better solution would be to further limit the grouping to 6 coverpoints per group – which would lead to 64 bit cross-products per covergroup that can be assembled into available BRAM resources efficiently. Besides making it possible to use a single unified BRAM that would greatly simplify coverage arbitration hardware, it also establishes a more effective

usecase for coverage map write-buffers disscussed in 3.4.3.

4.3 Coverage Evaluation

4.3.1 Motivation for Coverage

To motivate the need for coverage in verification, consider a real-world example in testing BlackParrot [91]. RISC-V Tests [6] is a suite of individual targeted functional test programs, a group of which tests the floating point unit (FPU) in RV64FD or RV32FD [129] compliant hardware implementations. All the FPU test cases in RISC-V Tests execute on BlackParrot without any failures. They also do not show any mismatches during *cosimulation* against a golden-reference ISA simulator, Dromajo[122]. However, relying on the passing test cases as an indication that the implemented FPU in BlackParrot is functionally correct would be imprudent. Case in point, the FPU precision bug described in Section 4.4. This bug goes undetected through the RISC-V Tests benchmark suite. And obtaining the coverage effected by the test cases identified the reason in retrospect to be uncovered branches in the hardware description by the programs of the benchmark.

Objectively evaluating coverage metrics could be tricky mainly because of the huge coverage state-space. Defining a coverage metric is context-dependant. For example, software testing often characterizes coverage of a test case based on number of lines of the software that were made to execute (code-coverage), or the number of different combinations of control statements exercised, etc. Often times, coverage is in the context of optimizing an objective. In our use case, that objective is in exercising as many functional states combinations of the hardware. Looking at coverage as an indication of the *progress* of verification, different coverage metrics highlight different frontiers of progress. Some research works have explored different coverage metrics, and evaluated their relevance and utilities [103, 69, 93]. Some of the metrics that can be objectively evaluated across automated coverage metrics are:

- Completeness in identifying coverpoints,

Coverage Metric		Coverspace (bits)
Mux Toggle Coverage	*if* statements	5849
	Ternary statements	1486
	case statements	196
	Total	**7531**
Mux Toggle Coverage + Post Processing	*if* statements	1399
	Ternary statements	810
	case statements	111
	Total	**2320**
Mux Toggle Coverage + Post Processing + Coverspace Reduction		**2190**
Case-Statement Coverage		196
Case-Statement Coverage + Post Processing		111

Table 4.4: Number of coverpoints identified for different coverage metrics, their composition, and improvements through post processing and coverspace reduction.

- Concision of the coverpoints for the same bug-discovering capacity, and,

- Time taken for coverage convergence, or saturation when appropriate. In some cases, the number of programs or total instructions emulated for coverage convergence could be more meaningful.

Because ZP Cosim only implements Mux-Toggle coverage, there is no reference or a baseline to compare the completeness of the metric against other coverage metrics.

The evaluation of the coverpoints identified by ZP Cosim's walker in the description of BlackParrot processor is compiled into Table 4.4. Post-processing is done through parse

scripts to reduce duplicates and reset-only coverpoints. Post-processing also identifies parameters-only and constant-only coverpoints (through parameter and constant substitution). Note that the UHDM elaboration also eliminates parameterized blocks that are not activated in the chosen parameter config. Additionally, note that the coverage statespace reduction of about 6% (130 bits) in BlackParrot may be significant in that the failing efforts needed for final sign-off can be avoided. Curiously, in a test evaluation of Chisel-lowered Verilog, we found that in the case of BOOM[137], case-statement coverage does not identify any coverpoints presumably because the Chisel compiler lowers the hardware description to synthesizable Verilog while also performing optimizations such as Dead Code Elimination (DCE), logic inference, and logic simplifications. One of the structural optimization is in reducing Case Statements in Chisel to a series of `if` statements in Verilog.

Note that a cross-product on the post-processed and reduced coverspace of 2190 bits would lead to a cross-coverage state-space of 2^{2190}, whereas exercising localized group-wise cross-products with $G = 10$ leads to a cross-coverage state-space of 219×2^{10} which is approximately 2^{212} times smaller.

4.3.2 Coverage over benchmarks

BlackParrot was evaluated with 300 randomly generated test programs with RISC-V DV [34], and 4 benchmark suites: RISC-V Tests [6] for targeted functional test cases, BEEBS [89] for its instruction distributions and its adaptability to bare metal execution, SPEC [7] for extremely long-running performance-related insights, and BP-Tests, an in-house developed suite of customized test cases for previously discovered bugs. BlackParrot is also capable of booting Linux, and so, the coverage effected by running Linux is also recorded. Figure 4.1 is a graph of toggle coverage achieved over some of the benchmarks as captioned. The black pixels correspond to coverpoints that have not been toggled, and the white pixel correspond to toggled coverpoints which are 88.01% of the total coverpoints. The inference is that there is still some space of the hardware functionality that remains to be tested, and that begs for further testing.

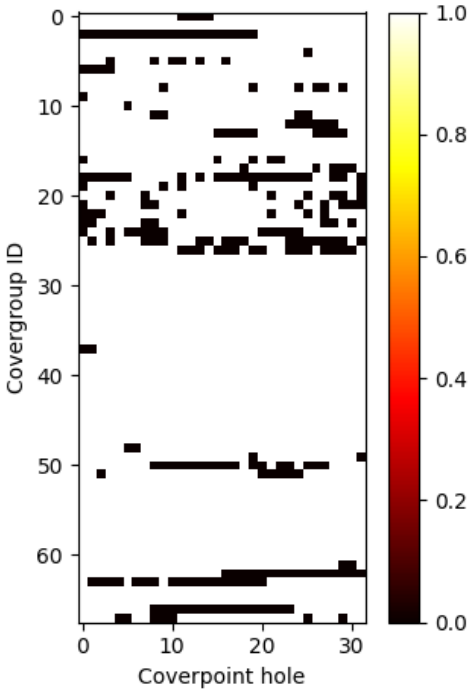
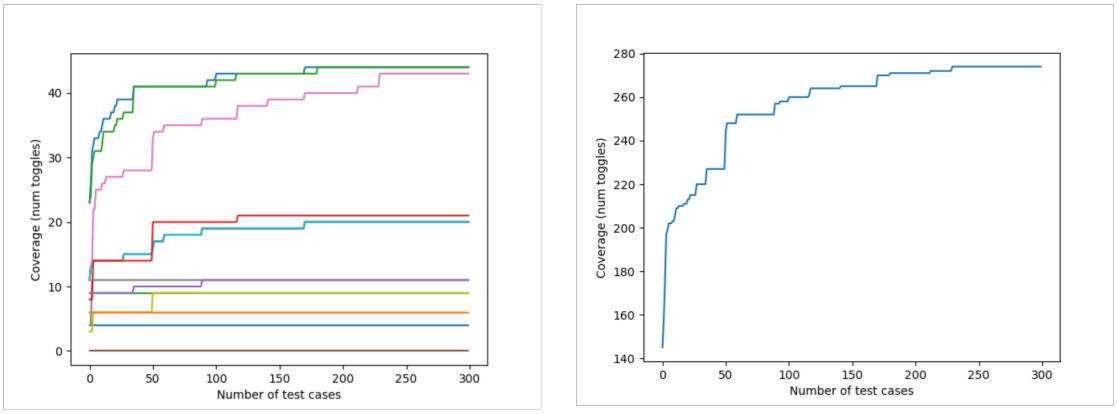


Figure 4.1: The coverage map indicating "coverage holes" after running the RISC-V Tests, BEEBS, and 4 of 9 SPEC 2017 benchmark programs. The Y-axis represents the covergroup IDs, and the X-axis represents the individual coverpoint toggles within the corresponding covergroups.

4.3.3 Coverage Convergence

For the Mux-Toggle coverage metric, we plot the coverage effected by 300 randomly generated programs on a coverspace of 600 bits or 60 covergroups each with 10 coverpoints in simulation, and the coverage convergence is plotted in Figure 4.3. As for the case-statment coverage metric, there are 46 `case` statements with unique condition expressions distributed between 1 bit wide to 6 bits wide, totalling to 151 bits[3]. A simulation run in Verilator with the same 300 randomly generated programs (generated by `riscv-dv`) is captured in the coverage graph

[3]We manually ignore the 32-bits instruction word in cases where it appears.



(a) Individual coverage effected by each of the groupings at G=10

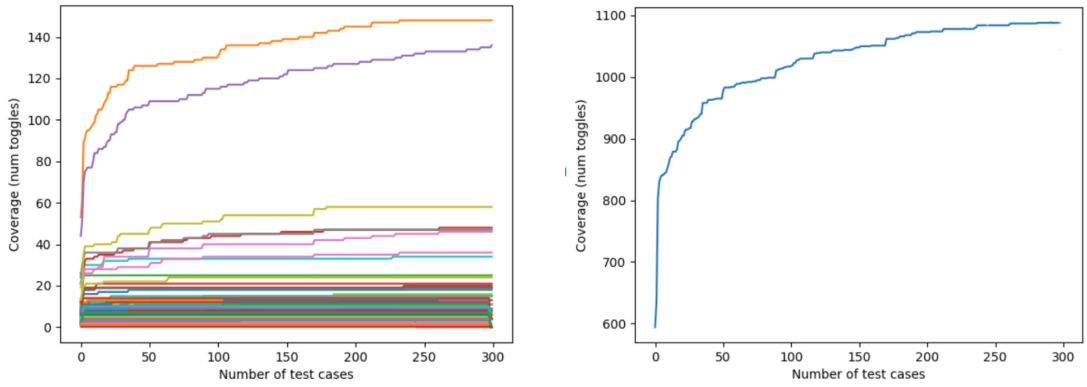
(b) Cumulative coverage effected in all the identified coverpoints

Figure 4.2: Case-statement coverage effected by RISCV-DV generated programs on Black-Parrot.

in Figure 4.2. The coverage effected is plotted both group-wise (according to ZP Cosim's grouping of coverpoints), and cumulatively.

One of the important observations is that coverage saturates early into program runs, following which there is infrequent and spaced out increments to coverages. ZP Cosim implements the coverage modules such that all of them can be read out sequentially. However, barring a few initial programs, most other times, just transferring individual coverpoint hits to the PS makes sense.

Another related observation (not from the graphs above) is that most of the initially hit coverpoints are consistently hit. This was confirmed by randomly resetting the coverage hits and restarting the coverage counts, and the convergence graph remains very similarly shaped. This leads to the conclusion that some coverpoints are very frequently and very easily hit – causing the large initial boost in coverage. And the rest of the coverpoints are very rarely (if at all), and very infrequently hit. This allows for very useful optimizations in



(a) Individual coverage effected by each of the groupings at G=10

(b) Cumulative coverage effected in all the identified coverpoints

Figure 4.3: Mux (select-signal) toggle coverage effected by RISCV-DV generated programs on BlackParrot.

the coverage infrastructure in the shell – mainly, the use of a unified write-buffer as there are rarely ever updates to the coverage hit memories. Even if conservatively, we maintain a shallow write-buffer, and drop write-updates whenever the write-buffers are full – which will be a lot of writes initially, eventually, most of the easily-reachable coverpoints will be hit, and moving closer towards convergence, inevitably reduces the number of writes to the coverage hit memories.

Ideally, the best way to collect coverage would be by running a small number of benchmark programs first – which hits most of the easily hittable coverpoints. When the coverage is stable, which is when the easy coverpoints are all hit, the updates become less frequent, and the PS can then rely on individual coverpoint hit updates from the shell to guide its fuzzer or any other form of analysis tool to learn what changed in the program that caused the increment in coverage, and try and mutate the cause of that with other stimuli to hit even more coverpoints in combinations.

4.3.4 Insights

ZP Cosim's implementation of Mux Toggle coverage has a few drawbacks:

Implicit Muxes The walker does not recognize implicit muxes during its identification of coverpoints in the input hardware design. Implicit muxes are those that are elaborated in the design itself, that logic inference[4] tools would otherwise interpret as and synthesize into muxes.

Coverage Aliasing The walker does not preclude coverage aliasing which is discussed in Section 3.1.9.

Cross-Coverage The implementation of grouped cross product (as opposed to total cross-product) on coverage may sacrifice useful coverage for being able to FPGA-accelerate coverage-extraction as otherwise the coverage infrastructure will not fit on the FPGA; this is discussed in Section 3.1.11.

Coverage Metric The Mux Toggle Coverage metric is also not a stand-alone metric. For example, expression coverage, sometimes also called condition coverage, is defined as the coverage over all the expression and subexpressions in the design description. There could be cases where an expression can evaluate the select signal of a mux, but the expression could be buggy while evaluating to both true and false incorrectly. Relying solely on the Mux Toggle coverage might not help discover the bug in the expression as fuzzers may not "work towards" discovering all evaluations of the expression. So there needs to be a combination of different coverage metrics for a more "complete" indication of coverage.

---

[4]Logic inference during ASIC or FPGA synthsis, is the mechanism of detecting and recognizing particular code patterns in the logic description that can be supplanted with standardized logic elements available in the library. For example, a code structure like `assign X = A & S | B & ~S` is inferred as a Mux with two wire inputs A and B, a wire select S, and a wire output O.

Simulation reproducibility In addition, there is the possibility of the offline simulation not matching emulation – in that the mismatches encountered during emulation may not be visible during simulation in Verilator or VCS. This could be because of some of the non-deterministic design elements in the implementation of ZP Cosim – clock gating, which depends on the PS dequeuing the shell FIFOs, and DDR response latencies, which depends on other contending DDR accesses such as from the PS itself. Deterministic replay in SoC verification is the ability to replay failing test vectors to observe the same sequence of hardware states and the failing condition. Without reproducibility, the value of bug finding rests entirely on how good the real time debuggability of emulated designs is, and even then it will be restrictive with the insight that can be had with replays.

In order to support complete emulation reproducibility, all not-deterministic behavior of the hardware and the FPGA support environment need to be made deterministic or eliminated. In ZP Cosim, there are two sources of non-determinism:

1. Clock-gate assertion signal depends on the rate of dequeuing of the commit FIFOs (any of the full signals assert the gate). The FIFOs are dequeued by the control program executing on the PS which has many non-deterministic agents such as uncached memory accesses among those by other background processes in the operating system, context switches by the operating systems, etc.

   ZP Cosim allows for a configurable commit FIFO depth – when set to 1, every instruction commit in BlackParrot asserts the gate on the clock to BlackParrot. Instruction commits are deterministic[5] with the next adjustment.

2. DRAM latencies: ZP Cosim can be modified to implement an additional gating assertion signal that is asserted at every DRAM request generated at the adapter interface, and deassert it when the corresponding DRAM response arrives. This

[5]When memory accesses are deterministic too.

way, what would have otherwise been a non-determinable latency of request, will be a deterministic 1 cycle latency form the processor's point-of-view.

There are also additional bug-discovery advantages to implementing this[6]

4.4 Practical Utility and Reliability

The true measure of any verification infrastructure is in discovering known bugs and its potential of discovering new bugs. ZP Cosim has supported the discovery of 4 **new** bugs in BlackParrot processor. Note that BlackParrot is silicon-validated, so the impact of the bugs discovered is high. The following subsections describe the bugs, their importance, and reasoning on why they were not found through previously exercised verification methodologies.

4.4.1 Bug 1: FPU Precision

BlackParrot implements both single and double precision Floating Point extensions of the RISC-V ISA. In the microarchitecture, however, BlackParrot only implements the double-precision FPU with modifiable control signals to execute single-precision operations. In the buggy state, a series of accumulations of single-precision values lose precision gradually because of how rounding was handled for both single and double precision registers similarly in BlackParrot. Because the placements of the guard and round bits differs in bit positions across the two precisions, there have to be dedicated implementions of rounding for single and double precision values.

This was discovered in the process of running the SPEC [89] benchmark suite, which runs long sequences of programs with higher probability of encountering accumulations into the

[6]Reproducible randomness can often be useful in a more robust closed-loop fuzzing. These could be in the form of injecting external interrupts, or asserting clock gates on various clock domains – such as modulating the physical execution characteristics like latencies of DRAM accesses. The quality of reproducibility is important here because any bug or a hardware malfunction or a misbehavior encountered during fuzzing, needs to be reproducible for later offline debugging and bug fixing. In ZP Cosim, this can be implemented with configurable (between 1 and N cycles) DRAM response latencies as seen at the AXI interface at BlackParrot. This is possible because any chosen latency is an addition to the latency of 1 clock cycle made possible by the additional clock gating assertion. A reproduction emulation would then imitate the exact gating assertion and deassertion sequence.

same register. This was made possible because of the orders of magnitude faster cosimulation speeds of ZP Cosim.

4.4.2 Bug 2: Negative Zero

BlackParrot's Floating Point Unit (FPU) implementation superimposes multiplication and the Fused Multiply Add (FMA) operations. The FMA unit is part of a translated, and slightly modified HardFloat [25] implementation (originally in Chisel). The FMA unit takes 3 parameterizable data inputs, A, B and C, and control signals specifying the rounding mode, the specific operation to be performed, among other things, and performs the operation below:

$$result = \pm A \times \pm B \pm C$$

For floating point multiplication, BlackParrot retains `A` and `B` and adds `C = +0`. However, according to the IEEE 754 Standard for Floating Point Arithmetic [14] that the Instruction Set Architecture adheres to, adding a `+0`, is not always an additive identity:

$$(-0) + (+0) = \begin{cases} -0 & if roundingmode = RDN \\ +0 & if roundingmode \neq RDN \end{cases}$$

Clearly, adding a `+0` to the result of the floating point multiplication can change the sign of the result when the rounding mode for the operation is set to anything other than RDN (Round Down). The bug in question is because of BlackParrot not accounting for this possibility. During cosimulation, instructions `fmul.d` and `fmul.s` which are floating point multiply instructions in double and single precision, respectively, that ideally should compute to a `-0` when the rounding mode is not RDN, end up computing to `+0`, mismatching with reference Dromajo simulation.

This bug was not encountered in RISC-V Tests or BEEBS, but numerous times in running the SPEC benchmarks. An important observation is that the average execution time of each the SPEC benchmarks programs is significantly higher than BEEBS and RISC-V Tests programs. Longer-running test programs have greater probability of hitting more coverpoints

or crosses. This led to further construction of a more complete sweep of special operand values (NaN, +inf, -inf, +0, -0, negative and positive fractions and integers), and rounding modes. This manual exercise of inferring possible interesting cases from a mismatching case, and constructing programs to exercise those interesting case methodically is something that an ideal fuzzer should be able to automate with properly guided mutations and program generations.

4.4.3 Bug 3: NaN-Boxing

In implementations that support both single and double precision floating point operations, and allow for shared register files, there is the issue of distinguishing precision. NaN-boxing is a requirement imposed by the ISA, where, a narrow `n`-bit value, where $n < FLEN$ and `FLEN` is the maximum supported width of the FPU Register File, need to be NaN-boxed for it to be valid, i.e., all upper $FLEN - n$ bits are set to 1. In our case, NaN boxing requires the implementation to set the upper 32-bits of a 64-bit register if the lower 32 bits are used to hold 32-bit floats. As an example, when a `flw fa4, 0(a4)` instruction is executed, the 32-bit data, say `0xdead_beef`, at the address `a4+0` is loaded into the lower 32-bits of fa4, while also setting the upper 32 bits, essentially resulting in fa4 holding `0xffff_ffff_dead_beef`. In implementations that do not implement correct NaN boxing, a non-boxed single-precision value will be considered as a NaN (from the way NaNs are encoded in 64-bit double-precision encoding). As a corollary, operations (double-precision, for example) performed on narrower (single-precision, for example) registers should compute to NaN. BlackParrot had a bug where this was not recognized as a case to NaN the result.

4.4.4 Bug 4: Integer to Floating Point Conversion

In the buggy state, during the execution of a `fcvt.s.w` which converts a signed 32 bit integer value to a single-precision floating point value, BlackParrot fails to recognize the sign of the integer, essentially executing `fcvt.s.wu` converting an unsigned integer instead of a signed integer.

This is not particularly a hard-to-find bug, but having the insight on the coverage holes left over by previous programs definitely helps. The emulation framework did not necessarily help with the discovery by itself, but hte automated coverage instrumentation contributed the insight.

4.5 Existing Work

4.5.1 Open-Source Alternatives

The work on Simulator Independent Coverage by Kevin Laeufer et. al. [80] carries a similar motivation as ZP Cosim – the need for automated coverage metrics that enable more accurate and customized coverage measurement. The work builds on FireSim, Fromajo and related projects [73, 137, 74], and demonstrates and assesses various automatic coverage metrics [93], and describes compiler support in Chisel hardware description language for extracting and synthesizing cover statements that estimate simulation-time coverage. They propose a common coverage format and a report generator that can be used to assemble and accumulate coverages across different simulator and emulator runs. However, the infrastructure here relies heavily on compiler support which adds significant engineering effort in porting to other high-level languages. The cover statements count the number of times a coverage target was hit on a test using a saturating counter, and so, is viable for limited number of coverage targets. In large designs, however, the automation may not scale.

ZP Cosim also demonstrates and assesses automatic coverage metrics, and relies on compiler support. However, the hardware description languages it supports, Verilog and SystemVerilog, are more popular and sometimes the only language that commercial chip designs are described in. Additionally, the coverage metrics as identified in Chisel, are not always relevant in the generated Verilog which adds to the instrumentation complexity; the Chisel compiler will also need to raise the coverage information back to Chisel for it to be interpretable in context of the high-level language, which adds overheads in compilation and coverage interpretation. The compiler support in ZP Cosim comes in the form of a walker

skeleton for the open-source Surelog-compiled Universal Hardware Data Model that can be used as is, or developed to implement custom coverage metrics. The walker, written in C++, walks a standardized SystemVerilog Object Model [13] through standard VPI APIs and is more adaptable in practice. In addition, the coverage extraction can be limited to key submodules needed to be tested by changing just one flag during the invocation of the walker which can enable localized coverage extraction. Moreover, unlike the implementation in comparison, ZP Cosim allows for a zero-footprint[7] instrumentation with SystemVerilog hierarchical scopes. As such, ZP Cosim's approach is more customizable and adaptable. The option of emulation on FPGA clusters makes ZP Cosim more easily scalable and emulation-friendly; the work in comparison also allows for FPGA cluster emulations on FireSim[73] instances, but the cost offsets are unclear due to their use of expensive servers and cloud FPGA infrastructures with expensive PCIE interfaces.

4.5.2 Commercial Alternatives

Among the industrial solutions, ImperasDV [3] matches all the feature set ZP Cosim provides. ImperasDV has its own reference ISA simulator, Imperas Instruction Set Simulation (ISS) and the riscvOVPsim which supports cosimulation against simulated and emulated hardware targets designed in Verilog/SystemVerilog. They provide mechanisms for functional coverage measurement – not the broader microarchitectural coverage like ZP Cosim does, and the testbench/harness for interfacing hardware targets. The also have a customizable random instruction test generator for a similarly closed-loop verification strategy as ZP Cosim. The main areas we hypothesize ZP Cosim would perform well is in the modifiability of the platform, cost of the overall verification infrastructure – which is mainly the cost of the inexpensive FPGA boards and the cost of operation, and a more extensive and pliable coverage measurement infrastructure.

[7]Zero footprint in submodules. Hierarchical scopes on variables allow wiring variables from deep within the module hierarchy into modules in the top-module without having to introduce any extraneous IO in the hierarchy. Of course, the coverage modules of ZP Cosim can be placed in a separate file, and wrapped together with the original "truly untouched" hardware description of the implementation under test.

Renode [19] defines cosimulation to mean a full system simulation where Renode simulates a part of the system – usually, the host processor which can be chosen from an arsenal of ARM and RISC-V implementations, and a "verilated" [8] user designed accelerator, peripheral IP, or processor. Renode supports communication with the verilated part via AXI or Wishbone [60] with simple user-defined software shims. The infrastructure is still evolving and open-sourced. Renode also provides a vast feature set that includes automated CI setups for testing soft cores and versatile prototyping of processor designs.

4.5.3 Alternative Components

Spike [8], like Dromajo, is an alternative C++ functional model for RISC-V ISA. Spike also provides support for cosimulation and serves as a golden-reference model. For application execution environment, Spike is often paired with RISC-V Proxy Kernel. Spike supports RV32GC (just like Dromajo) and extends support to Q (quad-precision floating point instructions) and V (vector instruction) extensions. Spike also provides parameterizable memory consistency models: Weak Memory Ordering (WMO) and Total Store Ordering (TSO), and supports the Debug specification for visibility into memory/register contents.

RISC-V Architectural Test Suite and (a broader) RISC-V Compatibility Framework [11] are minimum necessary test programs to be passed for licensing of hardware implementations with a corresponding RISC-V ISA. Passing here does not indicate that the implementation is fully compliant, but rather checks important aspects of the implementation and the specification matching. One of the goals of ZP Cosim is to generate minimal high-coverage test cases for checking microarchitectural compliance with the corresponding RISC-V ISA, and to discover bugs in the process. The two test suites in question provide a set of readily available tests for ISA-level compliance.

RISC-V Formal Verification Framework [41] is another alternative to functional cosimulation with RISC-V ISA simulators. It features a processor-independent formal description of

[8] Verilated peripherals are hardware peripherals that have been compiled into a functional C++ model that can be executed, usually with a user-customizable wrapper.

the RISC-V ISA, and set of formal testbenches to verify compliance. The setup costs include designing the RISC-V Formal Interface (RVFI) which is the verification-only scaffolding for interfacing with RISC-V Formal. This allows for a portability of the verification infrastructure. Though this may involve intrusive changes to the processor under test, a sequential equivalence check verifies the equivalence of the core with/out the RVFI modification, so the verification reliability is unaffected. RISC-V Formal uses the SymbiYosys formal verification front-end which helps in composing portable formal checks.

CONCLUSION

This thesis introduced ZP Cosim as an enabler towards the concept of iterative, self-contained processor verification. ZPD combines portable and configurable toolset, an FPGA shell, and a control program, enabling cosimulation and automated coverage extraction.

The toolset is comprised of automated, concision-driven coverpoint identification, and non-intrusive instrumentation of the processor-under-test. This toolset is completely open-source and portable due to the use of standard languages and tools.

The FPGA shell interfaces with the PUT (here, BlackParrot, a RISC-V processor) on one side, and provides a decoupled, standard AXI interface with an ARM based host processor on the other side. The shell allows BlackParrot to run asynchronously, and it does so without creating any back-pressure on the PUT, by instead clock-gating it and safely managing the clock crossings.

And the control program manages the execution on the PUT, extracts execution trace in real-time, cosimulates against a golden reference model (Dromajo RISC-V ISA simulator), and extracts coverage post-emulation. The coverage can be analyzed post emulation for completeness of testing, and can be utilized towards driving a coverage-guided fuzzing instance or manually targeting hitherto uncovered coverpoints.

Moreover, the cosimulation-based verification infrastructure can be scaled up to multiple FPGAs via FPGA-clusters to accelerate verification even further. And because of the use of inexpensive consumer-FPGA boards, the infrastructure is economical. In the process, the coverage effected by popular benchmarks and randomly generated programs were studied and the resulting gaps motivated the discovery of 4 key microarchitectural bugs in the silicon-validated BlackParrot processor.

BIBLIOGRAPHY

[1] 2022 Wilson Research Study – The Current Trends in ASIC Design Verification Technology. https://blogs.sw.siemens.com/podcasts/where-today-meets-tomorrow/2022-wilson-research-study-the-current-trends-in-verification-\technology-pt-2-of-2/. Accessed: 2010-09-30. 5

[2] Common Cells Repository. https://github.com/pulp-platform/common_cells. Accessed: 2010-09-30. 7

[3] ImperasDV. https://www.imperas.com/imperasdv. 62

[4] Must-have Verilog Systemverilog Modules. https://github.com/pConst/basic_verilog. Accessed: 2010-09-30. 7

[5] RISC-V Proxy Kernel. https://github.com/riscv-software-src/riscv-pk. 11

[6] RISC-V Tests. https://github.com/riscv-software-src/riscv-tests. Accessed: 2010-09-30. 50, 52

[7] SPEC CPU 2017 benchmark suite. https://www.spec.org/cpu2017/. Accessed: 2021-09-01. 52

[8] Spike RISC-V ISA Simulator. https://github.com/riscv-software-src/riscv-isa-sim. 63

[9] Zynq-Parrot GitHub Repository. https://github.com/black-parrot-hdk/zynq-parrot. 14, 28, 31

[10] Avnet Ultra96v2. https://www.xilinx.com/products/boards-and-kits/1-vad4rl.html, 2010. rev. version B4. 28

[11] RISC-V Architecture Test. https://github.com/riscv-non-isa/riscv-arch-test, 2010. rev. version B4. 63

[12] Synopsys VCS. https://www.synopsys.com/verification/simulation/vcs.html, 2010. rev. version B4. 27, 45

[13] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018. 17, 62

[14] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. 59

[15] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, D. Srikrishna, and M. Taylor. The RAW compiler project. In *Proceedings of the Second SUIF Compiler Workshop*, pages 21–23, 1997. vii

[16] Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Atieh Lotfi, Julian Puscar, Anuj Rao, Austin Rovinski, Loai Salem, Ningxiao Sun, Christopher Torng, Luis Vega, Bandhav Veluri, Xiaoyang Wang, Shaolin Xie, Chun Zhao, Ritchie Zhao, Christopher Batten, Ronald G. Dreslinski, Ian Galton, Rajesh K. Gupta, Patrick P. Mercier, Mani Srivastava, Michael Bedford Taylor, and Zhiru Zhang. Celerity: An Open Source RISC-V Tiered Accelerator Fabric. In *HOTCHIPS*, Aug 2017. vii

[17] Alric Althoff, Joseph McMahan, Luis Vega, Scott Davidson, Timothy Sherwood, Michael Taylor, and Ryan Kastner. Hiding Intermittant Information Leakage with Architectural Support for Blinking. In *International Symposium on Computer Architecture (ISCA)*, 2018. vii

[18] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020. 7

[19] AntMicro. Renode. `https://antmicro.com/platforms/renode/`. 63

[20] Manish Arora, Jack Sampson, Nathan Goulding-Hotta, Jonathan Babb, Ganesh Venkatesh, Michael Bedford Taylor, and Steven Swanson. Reducing the Energy Cost of Irregular Code Bases in Soft Processor Systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2011. vii

[21] Inc. Avnet. Ultra96-v2 single board computer hardware user's guide, version 1.3, May 2017. 31

[22] Jonathan Babb, Matthew Frank, Victor Lee, Elliot Waingold, Rajeev Barua, Michael Taylor, Jang Kim, Srikrishna Devabhaktuni, and Anant Agarwal. The Raw Benchmark

Suite: Computation Structures for General Purpose Computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 1997. vii

[23] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 1216–1225, New York, NY, USA, 2012. Association for Computing Machinery. 7

[24] B. Beresini, S. Ricketts, and M.B. Taylor. Unifying manycore and fpga processing with the RUSH architecture. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 22 –28, 2011. vii

[25] Berkeley. HardFloat implementation in Chisel. `https://github.com/ucb-bar/berkeley-hardfloat`. 59

[26] Vikram Bhatt, Nathan Goulding-Hotta, Qiaoshi Zheng, Jack Sampson, Steve Swanson, and Michael B. Taylor. Sichrome: Mobile web browsing in Hardware to save Energy . In *Dark Silicon Workshop, ISCA*, 2012. vii

[27] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. The essence of bluespec: A core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 243–257, New York, NY, USA, 2020. Association for Computing Machinery. 7

[28] Ajay Brahmakshatriya, Emily Furst, Victor Ying, Claire Hsu, Max Ruttenberg, Yunming Zhang, Tommy Jung, Dustin Richmond, Michael Taylor, Julian Shun, Mark Oskin, Daniel Sanchez, and Saman Amarasinghe. Taming the zoo: A unified graph compiler framework for novel architectures. In *ISCA*, 2021. vii

[29] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing, 2018. 12

[30] Cadence. Palladium Z2. `https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html`. 8

[31] Cadence. Protium X2. `https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/protium.html`. 8

[32] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *DAC*, 2021. vii

[33] Sadullah Canakci, Chathura Rajapaksha, Anoop Mysore Nataraja, Leila Delshadtehrani, Michael Taylor, Manuel Egele, and Ajay Joshi. Processorfuzz: Guiding processor fuzzing using control and status registers. *arXiv preprint arXiv:2209.01789*, 2022. 21, 45

[34] CHIPS Alliance. RISCV-DV. `https://github.com/chipsalliance/riscv-dv`, 2010. rev. version B4. 11, 44, 52

[35] Yuan-Mao Chueh. A Complete Open Source Network Stack For BlackParrot. Master's thesis, University of Washington, 2022. vii, 45

[36] Clifford Cummings. Simulation and Synthesis Techniques for Asynchronous FIFO Design. `http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf`. 35

[37] Alain Dargelas. Surelog SystemVerilog Compiler. `https://github.com/chipsalliance/Surelog`. 16, 17

[38] Alain Dargelas. Universal Hardware Data Model. `https://github.com/chipsalliance/UHDM`. 16, 17

[39] Scott Davidson, Shaolin Xie, Chris Torng, Khalid Al-Hawaj, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald Dreslinski, Christopher Batten, and Michael Bedford Taylor. The Celerity Open-Source 511-core RISC-V Tiered Accelerator Fabric. *Micro, IEEE*, Mar/Apr. 2018. vii

[40] C. Dawson, S.K. Pattanam, and D. Roberts. The verilog procedural interface for the verilog hardware description language. In *Proceedings. IEEE International Verilog HDL Conference*, pages 17–23, 1996. 17

[41] Symbiotic EDA. RISC-V Formal Verification Framework. `https://github.com/SymbioticEDA/riscv-formal`, 2010. rev. version B4. 63

[42] Hadi Esmaeilzadeh and Michael Bedford Taylor. Open Source Hardware: Stone Soups and Not Stone Satues, Please. In *SIGARCH Computer Architecture Today*, Dec 2017. vii

[43] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020. 12

[44] Harry D. Foster. Trends in functional verification: a 2014 industry study. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 48:1–48:6. ACM, 2015. 5

[45] Emily Furst. *Code Generation and Optimization of Graph Programs on a Manycore Architecture*. PhD thesis, University of Washington, 2021. vii

[46] S. Garcia, Donghwan Jeon, C. Louie, and M.B. Taylor. The Kremlin Oracle for Sequential Code Parallelization. *Micro, IEEE*, 32(4):42–53, July-Aug. 2012. vii

[47] Saturnino Garcia, Donghwan Jeon, Chris Louie, Sravanthi Kota Venkata, and Michael Bedford Taylor. Bridging the Parallelization Gap: Automating Parallelism Discovery and Planning. In *USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*, 2010. vii

[48] Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor. Kremlin: Rethinking and Rebooting gprof for the Multicore Age. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2011. vii

[49] Farzam Gilani. Zynq-Farm FPGA Cluster. `https://github.com/farzamgl/zynq-farm`. 31

[50] Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Jonathan Babb, Michael Taylor, and Steven Swanson. GreenDroid: A Mobile Application Processor for a Future of Dark Silicon. In *HOTCHIPS*, 2010. vii

[51] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *Micro, IEEE*, pages 86–95, March 2011. vii

[52] Nathan Goulding-Hotta. *Specialization as a Candle in the Dark Silicon Regime*. PhD thesis, University of California, San Diego, 2020. vii

[53] Nathan Goulding-Hotta, Jack Sampson, Qiaoshi Zheng, Vikram Bhatt, Steven Swanson, and Michael Taylor. GreenDroid: An Architecture for the Dark Silicon Age. In *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2012. vii

[54] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. DR-SNUCA: An energy-scalable dynamically partitioned cache. In *International Conference on Computer Design (ICCD)*, 2013. vii

[55] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. Time Cube: A Manycore Embedded Processor with Interference-Agnostic Progress Tracking. In *International Conference On Embedded Computer Systems: Architectures, Modeling And Simulation (SAMOS)*, 2013. vii

[56] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. Qualitytime: A simple online technique for quantifying multicore execution efficiency. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014. vii

[57] Scott Alan Hauck. *Multi-FPGA systems*. University of Washington, 1995. 42

[58] Byron Hawkins, Brian Demsky, and Michael Bedford Taylor. BlackBox: Lightweight Security Monitoring for COTS Binaries. In *Code Generation and Optimization*, 2016. vii

[59] Byron Hawkins, Brian Demsky, and Michael Bedford Taylor. A Runtime Approach to Security and Privacy. In *European Security and Privacy*, 2016. vii

[60] Richard Herveille. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. `https://antmicro.com/platforms/renode/`, 2010. rev. version B4. 63

[61] Steven Hoover and Ahmed Salman. Top-down transaction-level design with tl-verilog, 2018. 7

[62] Hu, Zhu, Taylor, and Cheng. FPGA Global Routing Architecture Optimization Using a Multicommodity Flow Approach . In *ICCD*, 2007. vii

[63] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021. 15, 21, 25

[64] Donghwan Jeon, Saturnino Garcia, Chris Louie, Sravanthi Kota Venkata, and Michael Bedford Taylor. Kremlin: Like gprof, but for Parallelization. In *Principles and Practice of Parallel Programming (PPoPP)*, 2011. vii

[65] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Kismet: Parallel Speedup Estimates for Serial Programs. In *Conference on Object-Oriented Programming, Systems, Language and Applications (OOPSLA)*, 2011. vii

[66] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Parkour: Parallel Speedup Estimates from Serial Code. In *USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*, 2011. vii

[67] Donghwan Jeon, Saturnino Garcia, and Michael Bedford Taylor. Skadu: Efficient Vector Shadow Memories for Poly-scopic Program Analysis. In *Conference on Code Generation and Optimization (CGO)*, 2013. vii

[68] L. John. Agile hardware design. *IEEE Micro*, 40(04):4–5, jul 2020. 1

[69] Jing-Yang Jou and Chien-Nan Liu. Coverage analysis techniques for hdl design validation. *IEEE Asia Pacific Conference on Chip Design Languages*, 01 1999. 50

[70] Dai Cheol Jung. Caches for Complex Open Source System-on-Chip Designs. Master's thesis, University of Washington, 2019. vii, 45

[71] Dai Cheol Jung, Scott Davidson, Chun Zhao, Dustin Richmond, and Michael Bedford Taylor. Ruche Networks: Wire-Maximal, No-Fuss NoCs. In *NOCS*, 2020. vii

[72] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3219–3236, Boston, MA, August 2022. USENIX Association. 2

[73] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 29–42, Piscataway, NJ, USA, 2018. IEEE Press. 8, 61, 62

[74] Sagar Karandikar, Albert Ou, Alon Amid, Howard Mao, Randy Katz, Borivoje Nikolić, and Krste Asanović. Fireperf: Fpga-accelerated full-system hardware/software performance profiling and co-design. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 715–731, New York, NY, USA, 2020. Association for Computing Machinery. 61

[75] Moein Khazraee. *Reducing the development cost of customized cloud infrastructure.* PhD thesis, University of California, San Diego, 2020. vii

[76] Moein Khazraee, Luis Vega, Ikuo Magaki, and Michael Taylor. Specializing a Planet's Computation: ASIC Clouds. *IEEE Micro*, May 2017. vii

[77] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Taylor. Moonwalk: NRE Optimization in ASIC Clouds or, accelerators will use old silicon. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017. vii

[78] Jason Kim, Michael B. Taylor, Jason Miller, and David Wentzlaff. Energy Characterization of a Tiled Architecture Processor with On-Chip Networks. In *International Symposium on Low Power Electronics and Design (ISLPED)*, August 2003. vii

[79] Sravanthi Kota Venkata, IkkJin Ahn, Donghwan Jeon, Anshuman Gupta, and Michael Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2009. vii

[80] Kevin Laeufer, Vighnesh Iyer, David Biancolin, Jonathan Bachrach, Borivoje Nikolić, and Koushik Sen. Simulator independent coverage for rtl hardware languages. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 606–615, New York, NY, USA, 2023. Association for Computing Machinery. 61

[81] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018. 15

[82] Ikuo Magaki, Moein Khazraee, Luis Vega, and Michael Taylor. ASIC Clouds: Specializing the Datacenter. In *International Symposium on Computer Architecture (ISCA)*, 2016. vii

[83] Subhendu Malakar. Deep reinforcement fuzzing, 2019. 12

[84] Sripathi Muralitharan. TinyParrot: An Integration-Optimized Linux-Capable Host Multicore. Master's thesis, University of Washington, 2021. vii, 45

[85] Anoop Mysore Nataraja. Zynq-Parrot-Dromajo. `https://github.com/black-parrot-hdk/zynq-parrot-dromajo`. 17

[86] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 544–558, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. 23

[87] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. Fuzzfactory: Domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. 12

[88] S. Pal, D. Park, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley, J. Beaumont, K. Chen, C. Chakrabarti, M. Taylor, T. Mudge, D. Blaauw, H. Kim, and R. Dreslinski. A 7.3 M Output Non-Zeros/J Sparse Matrix-Matrix Multiplication Accelerator using Memory Reconfiguration in 40 nm. In *Symposium on VLSI Circuits*, pages C150–C151, 2019. vii

[89] James Pallister, Simon Hollis, and Jeremy Bennett. BEEBS: Open benchmarks for energy measurements on embedded platforms. 08 2013. 52, 58

[90] D. Park, S. Pal, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley, J. Beaumont, K. Chen, C. Chakrabarti, M. B. Taylor, T. Mudge, D. Blaauw, H. Kim, and R. G. Dreslinski. A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix–Matrix Multiplication Accelerator. *IEEE Journal of Solid-State Circuits*, pages 933–944, April 2020. vii

[91] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, T. Guarino, A. Joshi, M. Oskin, and M. B. Taylor. BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro*, pages 93–102, Jul/Aug. 2020. vii, 28, 45, 50

[92] Daniel Petrisko, Chun Zhao, Scott Davidson, Paul Gao, Dustin Richmond, and Michael Bedford Taylor. NoC Symbiosis. In *NOCS*, 2020. vii

[93] Andrew Piziali. *Functional Verification Coverage Measurement and Analysis*. Springer Science & Business Media, 2007. 50, 61

[94] Shashank Vijaya Ranga. ParrotPiton and ZynqParrot: FPGA Enablements for the BlackParrot RISC-V Processor. Master's thesis, University of Washington, 2021. vii, 45

[95] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. Quickly generating diverse valid test inputs with reinforcement learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1410–1421, New York, NY, USA, 2020. Association for Computing Machinery. 12

[96] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond, Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski. A 1.4 GHz 695

Giga Risc-V Inst/s 496-Core Manycore Processor With Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS. In *2019 Symposium on VLSI Circuits*, pages C30–C31, 2019. vii

[97] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond, Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski. Evaluating Celerity: A 16-nm 695 Giga-RISC-V Instructions/s Manycore Processor With Synthesizable PLL. *IEEE Solid-State Circuits Letters*, 2(12):289–292, 2019. vii

[98] Jack Sampson, Manish Arora, Nathan Goulding-Hotta, Ganesh Venkatesh, Jonathan Babb, Vikram Bhatt, Michael Bedford Taylor, and Steven Swanson. An Evaluation of Selective Depipelining for FPGA-based Energy-Reducing Irregular Code Coprocessors. In *Conference on Field Programmable Logic and Applications (FPL)*, 2011. vii

[99] Jack Sampson, Ganesh Venkatesh, Nathan Goulding-Hotta, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor. Efficient Complex Operators for Irregular Codes. In *High Performance Computing Architecture (HPCA)*, 2011. vii

[100] Wilson Snyder. Verilator, a Verilog/Systemverilog simulator and compiler. `https://www.veripool.org/verilator/`, 2018. 6, 27, 45

[101] Steven Swanson and Michael Taylor. GreenDroid: Exploring the next evolution for smartphone application processors. In *IEEE Communications Magazine*, March 2011. vii

[102] Synopsys. Zebu Server 5. `https://www.synopsys.com/verification/emulation/zebu-server.html`. 8

[103] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design & Test of Computers*, 18(4):36–45, 2001. 15, 50

[104] MB Taylor, J Kim, J Miller, F Ghodrat, B Greenwald, P Johnson, W Lee, A Ma, N Shnidman, V Strumpen, et al. The raw processor-a scalable 32-bit fabric for embedded and general purpose computing. In *Proceedings of Hot Chips XIII*, 2001. vii

[105] Michael Taylor. *Tiled Microprocessors*. PhD thesis, Massachusetts Institute of Technology, 2007. vii

[106] Michael Taylor. A Landscape of the New Dark Silicon Design Regime. *Micro, IEEE*, Sept-Oct. 2013. vii

[107] Michael Taylor. A Landscape of the New Dark Silicon Design Regime. In *Design Automation and Test in Europe*, April 2014. vii

[108] Michael Taylor. The Evolution of Bitcoin Hardware. *Computer, IEEE*, Sept-Oct. 2017. vii

[109] Michael B. Taylor. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Design Automation Conference (DAC)*, 2012. vii

[110] Michael B. Taylor. Bitcoin and the Age of Bespoke Silicon. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2013. vii

[111] Michael B. Taylor. BaseJump STL: SystemVerilog needs a Standard Template Library for Hardware Design. In *Design Automation Conference*, June 2018. vii

[112] Michael B. Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. In *IEEE Micro*, March 2002. vii

[113] Michael B. Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Walter Lee, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Saman Amarasinghe, and Anant Agarwal. A 16-issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network. In *IEEE International Solid-State Circuits Conference (ISSCC)*, February 2003. vii

[114] Michael B. Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *International Symposium on High Performance Computer Architecture (HPCA)*, February 2003. vii

[115] Michael B. Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks. In *IEEE Transactions on Parallel and Distributed Systems*, February 2005. vii

[116] Michael B. Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, February 2005. vii

[117] Michael B. Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan

Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *International Symposium on Computer Architecture (ISCA)*, June 2004. vii

[118] Michael Bedford Taylor. Geocomputers and the Commercial Borg. In *SIGARCH Computer Architecture Today*, Dec 2017. vii

[119] Michael Bedford Taylor. Invited: Basejump stl: Systemverilog needs a standard template library for hardware design. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018. 7, 35, 41

[120] Michael Bedford Taylor. Your agile open source HW stinks (because it is not a system). In *ICCAD*, 2020. vii

[121] Michael Bedford Taylor, Luis Vega, Moein Khazraee, Ikuo Magaki, Scott Davidson, and Dustin Richmond. ASIC clouds: Specializing the datacenter for planet-scale applications. *CACM*, pages 103–109, 2020. vii

[122] Esperanto Technologies. Dromajo RISC-V ISA simulator. `https://github.com/chipsalliance/dromajo`, 2019. 26, 50

[123] Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. CortexSuite: A Synthetic Brain Benchmark Suite. In *International Symposium on Workload Characterization (IISWC)*, Oct. 2014. vii

[124] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3237–3254, Boston, MA, August 2022. USENIX Association. 2

[125] Luis Vega and Michael Bedford Taylor. RV-IOV: Tethering RISC-V Processors via Scalable I/O Virtualization . In *CARRV*, 2017. vii

[126] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010. vii

[127] Ganesh Venkatesh, John Sampson, Nathan Goulding, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. QsCores: Configurable Co-processors

to Trade Dark Silicon for Energy Efficiency in a Scalable Manner. In *International Symposium on Microarchitecture (MICRO)*, 2011. vii

[128] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to Software: Raw Machines. In *IEEE Computer*, September 1997. vii

[129] Andrew Waterman and RISC-V Foundation Krste Asanovi´c. The risc-v instruction set manual, volume i: User-level isa, document version 2.2, May 2017. 50

[130] Daniel Weber and Michael Schwarz. Cpu fuzzing: Automatic discovery of microarchitectural attacks. RuhrSec, 2023. 2

[131] Mark Wyse, Daniel Petrisko, Farzam Gilani, Yuan-Mao Chueh, Paul Gao, Dai Cheol Jung, Sripathi Muralitharan, Shashank Vijaya Ranga, Mark Oskin, and Michael Taylor. The blackparrot bedrock cache coherence system, 2022. 45

[132] Chenhao Xie, Xie Li, Yang Hu, Huwan Peng, Michael Taylor, and Shuaiwen Leon Song. Q-VR: System-level design for future mobile collaborative virtual reality. In *ASPLOS*, 2021. vii

[133] Shaolin Xie, Scott Davidson, Ikuo Magaki, Moein Khazraee, Luis Vega, Lu Zhang, and Michael B. Taylor. Extreme datacenter specialization for planet-scale computing: Asic clouds. In *ACM Sigops Operating System Review*, 2018. vii

[134] Florian Zaruba and Luca 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, 2019. 31, 34

[135] Karen Zee, Viktor Kuncak, Michael Taylor, and Martin C. Rinard. Runtime checking for program verification. In *RV*, 2007. vii

[136] Xingyao Zhang, Haojun Xia, Donglin Zhuang, Hao Sun, Xin Fu, Michael Taylor, and Shuaiwen Leon Song. η-LSTM: Co-designing highly-efficient large lstm training via exploiting memory-saving and architectural design opportunities. In *ISCA*, 2021. vii

[137] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. May 2020. 52, 61

[138] Ritchie Zhao, Chun Zhao, Shaolin Xie, Bandhav Veluri, Luis Vega, Christopher Torng, Ningxiao Sun, Austin Rovinski, Anuj Rao, Gai Liu, Paul Gao, Scott Davidson, Steve Dai, Aporva Amarnath, KhalidAl-Hawaj, Tutu Ajayi Christopher Batten, Ronald G. Dreslinski, Rajesh K.Gupta, Michael B.Taylor, and Zhiru Zhang. Celerity: An Open Source RISC-V Tiered Accelerator Fabric. In *7th RISC-V Workshop*, 2017. vii

[139] Qiaoshi Zheng, Nathan Goulding-Hotta, Scott Ricketts, Steven Swanson, Michael Bedford Taylor, and Jack Sampson. Exploring energy scalability in coprocessor-dominated architectures for dark silicon. *Transactions on Embedded Computing Systems (TECS)*, Mar 2014. vii

[140] Yi Zhu, Yuanfang Hu, Michael Taylor, and Chung-Kuan Cheng. Energy and switch area optimizations for FPGA global routing architectures. In *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, January 2009. vii

[141] Yi Zhu, Michael Taylor, Scott B. Baden, and Chung-Kuan Cheng. Advancing supercomputer performance through interconnection topology synthesis. In *International Conference on Computer-Aided Design (ICCAD)*, pages 555–558, 2008. vii

Appendix A

CODE REPOSITORIES

- BlackParrot is available as an open-source GitHub repository at `https://github.com/black-parrot/black-parrot`

- The hardware development kit for BlackParrot is maintained under `https://github.com/black-parrot-hdk`

  - Zynq-Parrot is the FPGA enablement for BlackParrot; it is located at `https://github.com/black-parrot-hdk/zynq-parrot`.

  - The thesis work is hosted by the ZP Cosim repository. It enables a complete closed-loop system verification, and is constantly evolving. It is available online, at `https://github.com/black-parrot-hdk/zynq-parrot-dromajo`. The repository README.md file details the procedure to set up the repository and exercise the verification framework described in the thesis.

- ZynqFarm is a private FPGA farm composed of 20 Ultra96v2 boards hosted by the Bespoke Silicon Group (BSG). It is available at `https://github.com/farzamgl/zynq-farm`. At the time of writing this thesis, the ZynqFarm is maintained by Farzam Gilani, a Ph.D. student at BSG. For research use, please contact Prof. Michael Taylor.