# Automated Testing, Verification and Repair of RTL Hardware Designs

by

Kevin Läufer

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Summer 2024

Automated Testing, Verification and Repair of RTL Hardware Designs

Abstract

Automated Testing, Verification and Repair of RTL Hardware Designs

by

Kevin Läufer

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Koushik Sen, Chair

All modern marvels of computer technology are built on microchips – grains of sand turned into computational components. Over more than five decades, we have seen rapid progress in computing performance, primarily thanks to semiconductor technology improvements that lowered power consumption and raised clock speeds. Recently, progress has stalled, and general-purpose hardware can no longer keep up with rising computational demands. Specialized hardware is the only way. Low-volume specialization is often not profitable, though, because chips are too expensive to design. While modern software developers have access to tools that allow them to innovate rapidly, hardware design tools are difficult to access and use. For this thesis, I investigated how ideas from modern software engineering can be applied to the hardware design domain.

Inspired by work on software compilers, I developed a new approach for adding coverage feedback to hardware designs such that many different simulators can be targeted with minimal effort. I built the RFUZZ tool, which uses coverage feedback to generate new test inputs, taking inspiration from work on mutational fuzz testing for software. I added support for formal verification to the open-source ChiselTest library, focusing on accessibility for new users. Finally, I designed the RTL-Repair tool, which automatically generates plausible repairs from a buggy hardware description and a failing test case through a combination of formal methods and simulation.

All four projects illustrate how we can improve the state-of-the-art hardware testing, verification, and debugging tools by integrating with the open-source ecosystem and applying a compiler engineering mindset. This approach has allowed me to quickly build superior coverage feedback and formal verification infrastructure for the new Chisel hardware language. It has also enabled the first hardware fuzzer, RFUZZ, which spawned a new line of research on hardware fuzzing, and the RTL-Repair tool, which provides correct repairs within seconds, several orders of magnitude faster than prior work.

Für meine Großeltern.

# Contents

# List of Figures

# List of Tables

## Acknowledgments

First and foremost, I would like to thank my advisor, Koushik Sen, who stuck with me for all seven years of my PhD. He discovered my potential and encouraged me to apply to PhD programs in the United States. He never stopped believing in me, giving me the stability to follow my own research agenda, even though – at times – I could not explain to him exactly what I was working on. Thank you, Koushik!

Jonathan Bachrach was another of my early supporters. He is always full of inspiring ideas, and he has helped me appreciate the artist's approach to research. I will always be motivated by his deeply held belief in the power of programming language designs to radically simplify common programming and design tasks.

Sanjit Seshia was my favorite teacher at Berkeley. With his insightful lectures and precise and compassionate way of answering student questions, Sanjit is one of my biggest role models for my own teaching style. My favorite time as a teaching assistant was working with Sarah Chasins, who was excellent at running her compilers class and always ensured that her TAs could grow their teaching skills without being overworked. Borivoje Nikolić always provided excellent, actionable feedback on paper drafts, even when I added him to the document out of the blue.

Most projects in this thesis would not have been possible without my great collaborators. Jack Koenig wrote the original FIRRTL compiler passes for rfuzz (Chapter 4), and Donggyu Kim almost magically enabled rfuzz to run on an FPGA within just a couple of days. David Biancolin ran all the FPGA benchmarks for the work on simulator independent coverage (Chapter 3) and wrote down the results for our paper. Brandon Fajardo and Abhik Ahuja worked with me for two years while they were undergraduates at Berkeley and provided me with an excellent study of all the bugs in the CirFix dataset, enabling me to design the RTL-Repair tool (Chapter 7).

Vighnesh Iyer was always the first person I would discuss research ideas with. He kept me sane, meeting with me weekly over Zoom while we were working from home because of the COVID-19 pandemic and later when I was working remotely from Cambridge. Tianrui Wei provided much-needed enthusiasm during the final years of my PhD. His insights into industry tools and his challenges to many of my ideas about tools and hardware language designs have resulted in many new research directions.

I want to express my gratitude to several members of the Chisel and greater hardware open-source community. Deborah Soung helped me understand how Chisel coverage is obtained in a commercial setting. Chick Markley was instrumental in adding cover support to the Treadle simulator. Tom Alcorn originally suggested adding a cover statement to FIRRTL, ultimately leading me to the idea presented in Chapter 3. Several members of the Chisel community helped lay the groundwork for the formal verification support described in Chapter 5: Tom Alcorn, Daniel Kasza, Jack Koenig, Deborah Soung, Chick Markley, Schuyler Eldridge, and Jiuyang Liu. I would also like to thank Claire Wolf for all she has done to advance the open-source Verilog ecosystem. Without the open-source yosys tool, many of my projects would have been impossible.

One of my biggest regrets about my PhD is that I never published a paper with any of the excellent researchers in my research group. I fondly remember Rohan Padhye explaining the intricate details of the Java ecosystem and the JVM to me. Caroline Lemieux and Rohan also worked on some fuzzing projects with me. They introduced me to the idea of keeping a second coverage map for valid inputs, which I used for the RFUZZ implementation. Caroline also showed me how to properly run evaluations of fuzz testing tools and the importance of quantitative data analysis for computer science research. Rafael Dutra graciously included me in his Quicksampler paper, introducing me to the paper-writing process shortly after I started my PhD. Rohan Bhavishi is one of the hardest-working people I know. He single-handedly saved our class project with monumental effort. We also spent many lunches supporting each other throughout grad school.

Benjamin Brock was the first person I befriended in Berkeley. I remember that on one of our daily runs, we talked about how cool it would be for me to also start a PhD in Berkeley, and then that actually happened when I got accepted about half a year later. Ben and Michael Dennis were my most important support system when – all of a sudden – I was applying to U.S. graduate schools with only a month left until the deadline. Without their feedback on my statement drafts and company, while working late at the lab, I could not have made it. Federico Mora was not only the foremost expert on SMT during my time at Berkeley but also an excellent friend. Backpacking provided me with important time away from research, and none of my trips would have been as much fun without this great group of people: Stanley Smith, Regina Eckert, Sarah Sterman, Justin Yim, and Emma Alexander. I also enjoyed several years of working with the CS graduate student association. In particular, I would like to thank Gabriel Fierro and Alon Amid for their leadership.

No research department works without staff who can help distracted professors and graduate students stay organized. The Aspire, Adept, and Slice labs that I was part of had an excellent team of staff who helped me succeed in my PhD: Roxana Infante, Tamille Chouteau, Ria Briggs, and Kostadin Ilov. On the department side, Shirley Salanio and Jean Nguyen helped me navigate the university bureaucracy and patiently answered my many questions.

I would like to thank my parents, who never cared about my grades as long as I put in the effort. They always supported me in my dreams, even if these took me far away from them, starting with a year of high school in the United States, leaving to study at RWTH Aachen, and then pursuing my PhD in Berkeley.

Finally, I would like to thank my soul mate. My wife has always believed in me, even when I was doubting that I would finish my PhD. She has brought me so much joy and supported me through the ups and downs of doing a PhD. I would also like to thank our two-year-old daughter for supporting me by taking regular naps and not missing me while she has fun at daycare.

# Chapter 1

# Introduction

The ability to execute programs at super-human speed and accuracy is the primary enabler of applied computer science. Most computation now happens on integrated digital chips comprising billions of transistors manufactured on a silicon wafer. Over the previous half-century, engineers have developed an elaborate stack of abstractions that enable us to build such complex systems [38, 125].

Many innovations have happened at the transistor and manufacturing level, making it possible to fit more transistors yearly on the same chip area, often for the same price. Gordon Moore at Intel observed in 1975 that the number of transistors doubled roughly every two years without an increase in cost, thus coining the term Moore's Law [34]. Dennard scaling enabled increased compute frequency for each semiconductor generation. By lowering the supply voltage, the power needed to switch a transistor is reduced, and thus, the switching frequency can be increased [40]. However, Dennard scaling does not consider that a minimum threshold voltage needs to be met for a transistor to switch reliably. Also, once the energy used to switch a transistor is sufficiently reduced, the so-called *leakage current*, consumed by any transistor powered on, becomes an essential factor. Thus, the frequencies at which digital VLSI designs could realistically be run stopped improving in the early 2000s [18]. The end of Dennard Scaling led to new power-aware designs that would selectively power down unused transistors or scale the operating frequency dynamically to stay within a defined power budget [139]. However, nowadays, in 2024, most experts agree that the pace of technological advancement has slowed down, the number of transistors is no longer increasing at the same speed, and the cost per transistor is not falling as rapidly anymore, thus spelling out the end of Moore's Law [140].

The vast improvements in transistor density and power consumption from the 1970s to the mid-2000s led to a focus on programmable general-purpose processor designs. There was little incentive to invest significant engineering resources into specialized chip designs if, by the time that design was finished, a programmable CPU manufactured in a new VLSI technology would be performant enough to make the specialized design obsolete. The electronic design automation (EDA) software that emerged is still heavily influenced by this singular focus. The tools are built for large teams of experts focused on heavily optimizing a single design. It is very much acceptable for a single EDA tool to require a dedicated engineer to configure and run. The focus on detailed optimizations means very little modularization and reuse.

In software engineering, however, development speed and time to market are paramount. Software development will always be different from hardware because it is feasible to update software, while changing a VLSI design after it has been fabricated is impossible. However, maturing VLSI technology and the need for specialization make it feasible and necessary to forgo some performance optimizations for lower engineering costs [1]. Small teams can use mature open-source compilers and libraries to build sophisticated software applications quickly. On the other hand, state-of-the-art industrial hardware design projects rely on proprietary simulators and synthesis tools of varying quality and feature support. Reuse mostly happens on a very coarse grain level of IP blocks. It often consists of copying source code into a project instead of linking to a library version that receives continuous updates.

The academic and open-source community has started to address some of these issues over the last 15 years. Work on new hardware construction languages (HCLs) has tried to answer the reuse problem by providing the means to implement flexible hardware generator libraries. Important examples are Chisel [7], Migen [20], Amaranth [150], SpinalHDL [116], Magma [142] and PyMTL [88]. The open-source yosys [153] synthesis tool enabled numerous new research tools that take advantage of its Verilog frontend and the RTL-IR circuit representation. Yosys can convert Verilog circuits into the much simpler botr2 [111] format, enabling a new generation of academic model checkers to work with standard circuits. Yosys is also an integral component of the OpenROAD project, which aims to provide a fully open-source flow for compiling register transfer level circuit descriptions into mask sets that can be fabricated [3]. An important research goal of OpenROAD is to allow for this conversion with minimal configuration and no human intervention. There is also a new generation of accelerator design languages (ADLs) which aim to raise the level of design abstraction such that abstract algorithm descriptions can be compiled down to efficient hardware [112, 124, 129].

New HCLs and ADLs offer a promising approach to making hardware designers more productive. However, just as important as creating the designs is to test that they work as expected. A recent study [56] reports that commercial semiconductor projects generally have more engineers working on verification than on design. These numbers reflect both the fact that testing and formal verification are paramount for a project's success, as well as the problem that current industry practices and tools require a lot of human engineering effort. In contrast to industry, a recent academic study found that many open-source hardware projects include few tests. Of the 50 most popular open-source projects for FPGAs on GitHub, "88% do not include test cases to reproduce bugs" [90]. This thesis presents four projects and tools that promise to make hardware verification engineers more productive,

[1]Intel, for example, decided to rely more on standard cells for their recent Lion Cove design, sacrificing maximum performance for faster time to market and reduced engineering costs. `https://chipsandcheese.com/2024/06/03/intels-lion-cove-architecture-preview/`

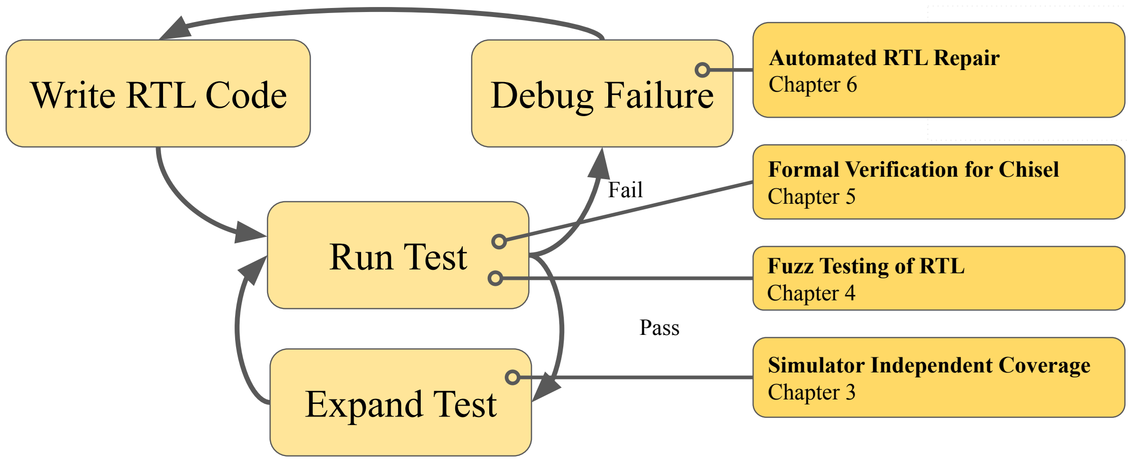


Figure 1.1: The agile RTL hardware development flow.

complementing the prior research on improving hardware design productivity.

Digital hardware is designed at many levels of abstraction: Functional simulators, RTL descriptions, and physical and standard cell design are all part of the process. However, most bugs are introduced at the register transfer level (RTL) [56]. RTL code is written by designers who use informal natural language descriptions of the expected functionality, which offers many opportunities to introduce incorrect behaviors. The RTL code precisely specifies the circuit's cycle-by-cycle behavior, and generating a mask set for tape-out is a mostly automated process, leaving little room for human-introduced errors. In the hardware context, verification refers to both testing an RTL design by executing it with a simulator (*dynamic verification*), as well as verifying a design with model checkers, proof assistants, or other formal tools (*formal verification*). The term *testing* is often used to talk about the process of finding manufacturing defects, which needs to take implementation details below the register transfer level into account. Throughout this thesis, which I write from a mixed software and hardware engineering perspective, I use RTL testing and dynamic verification interchangeably to try to bridge both worlds.

Through my research and my interactions with RTL designers in academia and industry, I have developed an idealized mental model of how RTL is or should be, written. This model is somewhat aspirational in treating verification as an integral part of development, similar to the test-driven development [11] methodology in software engineering. This lofty goal often falls short in practice because hardware designers feel they do not have time to write tests or because testing the whole system is too slow. Testing individual components is too difficult because of a lack of a cleanly defined interface or functionality. Nonetheless, this model has the advantage of being simple and helpful in illustrating verification ideas while being close enough to reality - especially for greenfield RTL projects - so they are not entirely irrelevant.

Figure 1.1 illustrates the *agile RTL hardware development flow.* The RTL designer writes

RTL code and then executes any available tests. If a test fails, they will have to look into the failure, figure out what is going wrong, and then update the RTL design to fix a bug. If all tests pass, this might mean that the RTL design is done and working as expected. However, it might also indicate insufficient testing, which means the engineer must expand the test suite. Throughout this thesis, I present four tools that promise to address the testing and debugging experience for RTL designers.

When RTL engineers write tests that simulate their design under test, they want to measure how thoroughly the existing tests cover the design functionality and identify parts of the design that require further testing. Coverage metrics quantify how often different features of a design are executed. Simple automated metrics check how many lines in the RTL source code are executed, how many states in a finite state machine are visited, or how many bits in a signal are toggled during a test's execution. More sophisticated functional coverage metrics require user input to define high-level features that must be covered. An example would be if the RTL designer writes code to capture how often the RTL implementation of a cache component executes a particular memory transaction.

While these coverage metrics are helpful, well known, and well supported in commercial SystemVerilog simulators, none of the simulators created for new hardware construction languages like Chisel supported them, and there were not enough engineers to painstakingly implement support for each metric in each simulator individually. Instead, I developed a new approach where coverage metrics are implemented only once as instrumentation passes in the FIRRTL compiler, which processes every Chisel-generated circuit. These passes generate synthesizable hardware, which simulators already had to support, and one new `cover` statement construct. I carefully designed this statement to be easy to implement across various simulators – from interpreters to FPGA-accelerated simulation – while being powerful enough to represent all standard coverage metrics. Chapter 3 shows how different coverage metrics and simulator support were implemented and that performance overhead is generally negligible.

Traditionally, engineers use the feedback from coverage metrics to extend and enhance their test suite manually. Would it be possible to use the coverage feedback to generate new test inputs directly, leading to greater coverage? I developed such a tool based on coverage-directed mutational fuzz testing, which had previously only been applied to finding security vulnerabilities and other bugs in software projects. Chapter 4 details the necessary steps to adapt the idea to hardware. I define a new coverage feedback metric that is easy to instrument, a way of mapping fuzzer-produced inputs to the cyclic execution of digital RTL circuits. I also demonstrate a tool architecture and isolation techniques that make it feasible to fuzz test designs running on an FPGA.

Simulation-based tests are easy to set up, and the required tools are widely available. However, they suffer from the fact that it is infeasible for all but the simplest circuits to explore all possible inputs exhaustively. On the other hand, formal verification promises to prove properties of a given design for all possible inputs and execution traces. However, formal verification tools have a reputation for requiring extensive background knowledge and being difficult to use. In Chapter 5, I detail my work on integrating bounded model checking into the ChiselTest library, making this powerful capability available to all Chisel users. This support was first released with ChiselTest 0.5.0 in 2021 and has since been used to write exhaustive tests for components from the Chisel standard library.

Finding a bug in the RTL design is only half the work. Once designers are faced with a failing test case, they still need to invest countless hours in analyzing the failure and developing a fix for the design. Automatic program repair promises to automate that task fully. However, results from a prior attempt to build such a tool for RTL designs were disappointing. The genetic algorithm-based tool would take minutes or hours to come up with a plausible repair, and many of these repairs would actually introduce new subtle bugs where the design would pass simulation, but the circuit generated from the high-level description would be buggy. Chapter 7 describes my RTL-Repair tool, which uses a novel repair algorithm based on ideas from bounded model checking to quickly (within seconds) generate repairs with a low false-positive rate. Since repairs are generated very quickly and with minimal changes, RTL-Repair has the potential to become an integral part of an RTL designer's workflow.

This thesis presents four tools for automated testing, verification, and repair of RTL hardware designs. All tools take advantage of and contribute back to the emerging new generation of open-source tools for chip design. While the majority of tools I built during the seven years of my PhD will not be successful in their own right, I believe they form a blueprint for how we can innovate in this new landscape of open EDA tools. We need to question and re-build the fundamentals of EDA software instead of trying to perform research on top of commercial tools, which we cannot modify or introspect. I could not build RTL-Repair without the formal verification library I built for ChiselTest and the yosys synthesis tool from the open-source community. The FIRRTL compiler and the open-source Verilator simulator enabled me to build the high-performance rfuzz fuzzing tool. A commercial simulator would have made the task much more difficult.

# Chapter 2

# Background

This chapter introduces basic concepts, tools, and methodologies for developing digital hardware. We define the register transfer level (RTL) of abstraction used by all hardware designs discussed in this thesis. All four tools from this thesis ingest descriptions of hardware designs expressed in a hardware description language (HDL) or hardware construction language (HCL). Thus, we provide context on both concepts. We then discuss in detail how bugs in RTL designs are found through simulation-based dynamic verification and formal verification.

## 2.1 Register Transfer Level (RTL) Circuit Design

Each chip design undergoes multiple levels of abstraction, eventually resulting in a set of photolithography masks used to fabricate the chip on a wafer. Each level of abstraction reduces the designer's cognitive load by limiting the degrees of freedom and the number of physical phenomena that need to be considered. Engineers always designed chips at various levels of abstraction, simulating a CPU design in software before drawing schematics and later polygons for photolithography masks. However, nowadays, the formerly painstaking and error-prone process of manually translating between abstraction layers has been mostly automated with software [38, 125]. Design still happens at all levels of abstraction. Still, most circuits are now designed at a high abstraction level and then automatically mapped to small primitives designed at a lower level of abstraction. For example, a high-level CPU description gets mapped to adders, flip-flops, SRAM memories, and other standard cells. Some common levels of abstraction are functional models, register transfer models, which include cycle accurate timing, gate-level models with gate delay information, gate-level netlists, which use standard cells of a particular VLSI process, circuit model simulations (SPICE), and mask sets.

In this thesis, we exclusively work with digital, synchronous circuits. Digital means that every wire in our resulting circuit carries either a zero or one value encoded as a voltage above or below a defined threshold. The voltage will fluctuate during computation, but the wire should carry a clear digital value when looking at converged instead of transient states. Synchronous means that state elements like registers and memories are all updated together with one global clock event. This implies that the slowest path for signal propagation through the circuit will determine the maximum frequency at which we can update our state elements. One single path determining the maximum frequency, a significant factor in computation speed, is a major downside that can be mitigated through various approaches, such as automated or manual retiming. The advantage of the synchronous approach is that fully automated tools exist to go from a high-level (RTL) description to a working circuit. Static timing analysis tools analyze the gate-level graph to ensure that no timing violations can happen. While asynchronous circuit design can mitigate the problem of the longest path determining computation speed, designing them requires much more manual effort, and often, the trade-offs are not worth it.

We define register transfer level (RTL) to mean that the designer explicitly defines each state element, like registers and memories. In our model, non-state, i.e., combinational circuit elements, do not carry delay information; outputs change instantaneously with the inputs. We can thus define each output bit as a boolean function of all input and state bits in the circuit. Circuit state elements are updated synchronously whenever a clock event happens. Each state bit's value after the clock event – the *next* value – is defined by a boolean function of all input and state bits before the clock event. Instead of working with boolean variables and operators directly, we often define *output* and *next* functions in terms of a finite bit-vector logic that simplifies our expressions while maintaining a straightforward lowering to boolean functions. This lowering is often called *bit-blasting*.

There are many ways of encoding the functions for the *next* value of a state element or the value of a circuit *output*. This thesis broadly distinguishes *structural* and *behavioral* descriptions. A *structural* description generally consists of a directed graph in which each node represents a combinational circuit element like a logic gate or adder, and edges denote connections between outputs and inputs and outputs of these elements. This description closely matches the resulting circuit of standard cells. It is also akin to the concept of a dataflow graph in the compiler literature. A *behavioral* description, on the other hand, is essentially a pure function that performs a bounded amount of computation over the function inputs and yields the output value as a result. Please note that the term *behavioral* in particular has carried various definitions in the literature, so our definition might not always agree with how other texts and researchers use the term.

## Converting Behavioral to Structural RTL

We can lower a *behavioral* function into a *structural* dataflow graph through symbolic execution where each function argument is initialized to a symbol; computations are applied symbolically to build up the data flow graph, and all possible branches through the program are explored. Branches are integrated into the data flow graph as If-Then-Else (ITE) expressions or multiplexers. ITE is the name used in the software domain; 2-input multiplexers are muxes that fulfill the same function in the hardware domain. Figure 2.1 shows

```
module Counter(
  input clock, input reset, input enable,
  output reg [3:0] count, output reg overflow
);
always_ff @(posedge clock) begin
  if(reset) begin
    count <= 'd0;
    overflow <= 1'b1;
  end else begin
    if(enable) begin
      count <= count + 'd1;
    end
    if(count == 4'b1111) begin
      overflow <= 1'b1;
    end
  end
end
endmodule
```

(a) Behavioral RTL description

```
module Counter(
  input clock, input reset, input enable,
  output reg [3:0] count, output reg overflow
);
  // wire declarations elided ...
  // registers
  always @(posedge clock)
    count <= count_next;
  always @(posedge clock)
    overflow <= overflow_next;

  // structural combinational logic
  assign count_plus_1 = count + 32'd1;
  assign count_is_15 = count == 4'hf;
  assign count_mux = enable ? count_plus_1[3:0] : count;
  assign count_next = reset ? 4'h0 : count_mux;
  assign overflow_mux = count_is_15 ? 1'h1 : overflow;
  assign overflow_next = reset ? 1'h1 : overflow_mux;
endmodule
```

(b) Structural RTL description

Figure 2.1: Lowering a counter from a behavioral to a structural RTL description.

```
module FalsePath(input a, input b, output logic [2:0] c);
logic [2:0] d;
always_comb begin
  if(a) d = 'd0;
  else  d = 'd1;
  if(a & d) c = 'd2;
  else      c = 'd3;
end
endmodule
```

(a) Behavioral RTL description. The assignment on the highlighted line is unreachable.

```
module FalsePath(input a, input b, output logic [2:0] c);
  assign c = (a & (a ?  3'h0 : 3'h1)) ?  3'h2 : 3'h3;
endmodule
```

(b) Structural RTL description

```
module FalsePath(input a, input b, output logic [2:0] c);
  assign c = 3'h3;
endmodule
```

(c) Structural RTL description after Optimization

Figure 2.2: Lowering of a behavioral design with an unreachable branch using yosys [153]. Naive symbolic execution creates a mux that includes the unreachable path condition `a & d`. Optimization can discard the false path.

the *behavioral* description of a simple counter circuit as well as the *structural* description of the same circuit. The open-source synthesis tool yosys [153] was used to parse the *behavioral* Verilog and to lower it into a *structural* version using the `proc` command. The resulting Verilog was then cleaned up manually to reduce the number of intermediate signals and to provide better names for signals generated by yosys. We see how every register is connected to two muxes, which reflects the three possible values they might transition to. The count register can either be reset to zero, updated with the sum of its prior value and one, or stay the same, which is expressed by assigning the old value.

There are two major challenges when converting a *behavioral* to a *structural* description: (1) naive approaches will include false paths with mutually exclusive path conditions, and (2) finding correct bounds for unrolling loop is hard. False paths are generally avoided through heuristics that let hardware compilers filter out trivially unreachable paths. This is often sufficient since behavioral hardware descriptions are often much simpler than general software programs. Figure 2.2 shows an example where the yosys synthesis tool includes a false path when lowering from behavioral to structural RTL (using the `proc` command). The false path is later optimized away through the `synth` command. The issue of bounding loops can be avoided by not allowing loops, like in the Chisel language [7], or by requiring that loop bounds must only depend on compile time known values like in the synthesizable subset of SystemVerilog [62].

## 2.2 Hardware Description Languages

The concept and challenges of register transfer level design are generally independent of the concrete hardware language that implements them. While countless languages have more narrowly focused on register transfer modeling, the industry's most popular hardware description languages are SystemVerilog [62] and VHDL [64]. Both languages support modeling circuits across several levels of abstraction, from behavioral RTL down to analog components and precise circuit delays. The underlying execution model of SystemVerilog and VHDL is based on parallel processes that emit and are scheduled in response to events such as value changes. The event-based model enables developers to simulate a wider variety of circuit components; however, efficiently mapping arbitrary circuit descriptions to an actual circuit implementation is essentially impossible. Thus, while simulators generally support all language constructs, other tools, like circuit synthesis or formal verification tools, work with much smaller *synthesizable subset* of the language [61, 38]. This thesis uses SystemVerilog and its predecessor language, Verilog, in various examples, such as Figure 2.1 and 2.2 in this chapter. A paper on "Verilog HDL and Its Ancestors and Descendants" [50] contains a more thorough historical perspective on the SystemVerilog language.

**Synthesizability.** Not all simulation constructs have a mapping to actual hardware, which leads to the definition of a synthesizable subset of the language [61, 135]. The mix of simulation language and automated translation can complicate hardware design: Circuits that seem to work well in simulation might fail to synthesize. A much more severe problem is synthesis-simulation mismatch, where a design is quietly accepted by the synthesis tool, but the resulting hardware behaves differently from the high-level HDL description [102]. Standard approaches to detect simulation-synthesis mismatch are combinational equivalence checking [77], which attempts to prove equivalence between the high-level RTL and the low-level netlist and gate-level simulations [50].

**X-Propagation.** In a SystemVerilog program, most values are 4-state bit-vectors: each bit can take on a value of 0, 1, Z, or X. The Z value is used to model tri-state buses. The X value is used to model unknown values. For example, these can originate from uninitialized state variables, out-of-bounds reads, unconnected signals, or explicit assignments of a signal to X [144]. Simulation with X values could be considered abstract interpretation since an X can stand for 0 and 1. However, in SystemVerilog, execution with X values is neither sound nor complete, meaning that for some computations with X, the result is over-approximated, and for others, it is under-approximated. Over-approximation, also known as X-optimism, can lead to a mismatch between the 4-state simulation and the 2-state circuit generated by the synthesis tool. X-propagation is thus a common source of synthesis-simulation mismatch.

## Shortcomings of Traditional HDLs

While SystemVerilog remains one of the most widely used languages in the industry, it has several significant problems: (1) language complexity, (2) semantic gap between event-driven execution model and circuits, and (3) lack of powerful meta-programming. The following paragraphs describe each problem in detail.

The complexity of language means that as of 2024, 19 years after IEEE first standardized it, there are no open-source tools that fully support the SystemVerilog standard [1]. Also, while hard data is difficult to come by, industry tools are also known to be lacking in full standard compliance, leading each semiconductor company to define its own subset of SystemVerilog that is safe to use with all tools. Static analysis tools called linters are then employed to ensure all code is written in the approved subset. In software development, on the other hand, many modern programming languages [2] feature only a single official frontend which defines all the languages semantics and thus all language features can be used and should work across development setups. In the past, older languages like C and C++ have struggled with similar problems to SystemVerilog, where different compilers would implement different versions of the language. Lots of standardization work and consolidation of compilers have mostly solved this problem. Currently, only three remaining C++ compilers follow the standard: Clang, GCC, and Microsoft Visual C++. ARM, IBM, and Intel abandoned their custom C++ frontends and now use Clang to process the input to their proprietary compilers [101, 35, 33].

The flexibility of the event-driven execution model makes it challenging to implement tools that need to extract a circuit view of a design. This includes circuit synthesis tools that lower a SystemVerilog description to a gate-level netlist, most formal verification tools, and circuit instrumentation tools for scan-chain insertion. While defining a synthesizable subset helps, it can be difficult for SystemVerilog users to understand which features are synthesizable. Even circuit descriptions that stay within the synthesizable subset can experience simulation synthesis mismatch. Strict linting rules can mitigate this problem. However, linters need to be purchased and configured.

While SystemVerilog features powerful circuit modeling capabilities, its features to generate circuit descriptions programmatically remain limited. Circuits can be parameterized using SystemVerilog parameters, generate-for and generate-if blocks, and compile-time evaluated loops and if-statements. These features are often used to set the bit widths of signals and configure optional features of circuit components. However, there are two general problems: (1) some approaches, like relying on partial evaluation at compile-time to generate hardware, depend on the sophistication of the particular compiler and thus are not guaranteed to be portable (2) more sophisticated generators that need to read in files or process information with custom data-structures are hard or impossible to implement with SystemVerilog. The

[1]The slang parser claims full SystemVerilog support, however, parsing is only a small step and - for example - the most popular open-source SystemVerilog simulator Verilator only supports a small subset of the language.

[2]Examples of modern programming languages with one a single *official* compiler are Rust, Go, and Swift.

standard approach for generating networks on chips or bus interconnects is thus to write a program in a general-purpose programming language like Perl or Python, which then generates SystemVerilog source code. Producing SystemVerilog strings directly, without going through a robust abstraction, can be error-prone and hard to debug, though.

These issues make it difficult to create reusable hardware components that can be shared across companies and teams. The language complexity also makes it very difficult to build new, innovative developer tools without a large team of engineers. Basic RTL descriptions are very simple, but SystemVerilog has evolved to contain a lot of accidental complexity [21], which must be dealt with to build reliable tools. Academic research is still feasible by focusing on a subset of SystemVerilog features required for the benchmarks used in a particular paper. However, none of these tools ever make it into the hands of end users since supporting arbitrary SystemVerilog designs is too difficult.

## 2.3 Hardware Construction Languages

While the languages discussed in the previous section focus on *describing* hardware behavior, a new generation of so-called hardware construction languages (HCLs) instead shifts the focus to writing reusable *hardware generators*. With ever-increasing SoC complexity, many designers aim to write RTL generators that can be extensively parameterized and reused. A prime example is the RocketChip SoC generator, which essentially takes in a list of devices to instantiate (e.g., cores, peripherals, accelerators) and automatically generates an RTL implementation of interconnects and device instantiations [6].

Hardware construction languages (HCLs), a term coined by the Chisel paper [7], provide a simple hardware description language that is in a general-purpose programming language that allows for powerful meta-programming. A Chisel generator is a Scala [113] program, which uses various functions from the Chisel package to build up a circuit description while executing. This process is called *elaboration*. Chisel is *deeply embedded* in the Scala host language since it builds up the circuit representation in memory [51].

The main contrast to previous approaches of generating, e.g., Verilog from a Perl script, is that the RTL constructs are not just strings but native objects in the host language, leading to better type safety and maintainability. Many HCLs are designed to make non-parameterized circuits look like they were written in a regular hardware description language. Chisel, for example, provides a `when` branch construct and assignment operators that work similarly to non-blocking assignments in Verilog.

Other HCLs are migen [20], which had its first commit in 2011, shortly after Chisel development had started and before the first paper on Chisel was published. Migen is embedded in Python. Its successor, which was rewritten from scratch, is Amaranth [150], which, among other things, generates hierarchical RTL descriptions. Migen would place all circuitry in a single module, making the output harder for humans to read and for backend tools to process. Other well-known HCLs are Magma [142] and PyMTL [88, 69], both of which are embedded in Python, as well as SpinalHDL [116] which is embedded in Scala.

```scala
class Counter extends Module {
  val enable = IO(Input(Bool()))
  val count = IO(Output(UInt(4.W)))
  val overflow = IO(Output(Bool()))

  val count_reg = RegInit(0.U(4.W))
  count := count_reg
  val overflow_reg = RegInit(false.B)
  overflow := overflow_reg

  when(enable) {
    count_reg := count_reg + 1.U
  }
  when(count === "b1111".U) {
    overflow_reg := true.B
  }
}
```

(a) Behavioral Chisel description

```
module Counter :
  // wire declarations elided ...
  // registers
  reg count_reg : UInt<4>, clock with :
    reset => (UInt(0), count_reg)
  reg overflow_reg : UInt<1>, clock with :
    reset => (UInt(0), overflow_reg)

  // structural combinational logic
  node count_plus_1 = add(count_reg, UInt(1))
  node count_is_15 = eq(count, UInt(15))
  node count_mux = mux(enable, tail(count_plus_1, 1), count_reg)
  count_reg <= mux(reset, UInt(0), count_mux)
  node overflow_mux = mux(count_is_15, UInt(1), overflow_reg)
  overflow_reg <= mux(reset, UInt(0), overflow_mux)
```

(b) Structural LoFIRRTL description

Figure 2.3: The counter from Figure 2.1 expressed in behavioral Chisel and its structural FIRRTL description.

## 2.4 RTL Intermediate Representations and Compilers

We first need to compile the Scala source code of a Chisel generator into Java byte code. Then, we execute the resulting binary, which uses the Chisel library to generate a circuit description. Some HCLs are implemented in a host language that does not need to be compiled, so we can skip the compilation step. However, the *elaboration* phase in which the generator is executed to create a circuit description is common to all HCL implementations. After *elaboration*, we generally want to obtain a description of the generated circuit that can be simulated, formally verified, or mapped to an FPGA or ASIC implementation. Commonly, HCLs produce an RTL description of the circuit in a subset of the Verilog or SystemVerilog language. Verilog is used as an interchange format since it is supported by virtually all open-source and commercial backend tools.

Some HCLs directly generate a Verilog description during *elaboration*. However, many HCLs feature an intermediate representation (IR) that represents the circuit after *elaboration* and a compiler that converts the IR to a Verilog output. Chisel used to generate Verilog directly, but since the Chisel 3 release, it utilizes the FIRRTL IR and compiler to perform the lowering from high-level, Chisel-like IR into a normalized structural circuit representation [67]. The low-level representation can then be exported into a subset of Verilog that was chosen as a common subset supported by the majority of backend tools. The introduction of a compiler simplifies the conversion from Chisel circuit constructs to an equivalent description in Verilog. The compiler implements type-checking, width inference, lowering of behavioral constructs and advanced data types, and some simple optimizations as individual passes. It is generally easier to correctly implement these individual passes than a single monolithic conversion to Verilog during elaboration. Other HCLs also feature an IR similar to FIRRTL [142, 150].

When I started working on this thesis in 2017, the FIRRTL compiler was under heavy development, and Chisel 3, the first version to rely on a compiler, was about to be released. While the most crucial advantage of the new architecture was to make correct Verilog generation more reliable and easier to maintain, Jack Koenig and Adam Izraelevitz were already looking for other applications that would benefit from this new compiler. I quickly picked up my colleagues' enthusiasm, and finally, three of the four projects I present in this thesis heavily use the FIRRTL compiler. For more details on the Chisel and FIRTL, I recommend reading Adam's PhD thesis [66].

The fuzzing framework I present in Chapter 4 features several new FIRRTL passes to collect coverage information and implement isolation techniques that allow us to quickly reset the design on an FPGA between running different fuzzer-generated inputs. Chapter 3 generalizes the coverage instrumentation approach using the newly developed `cover` construct to implement various coverage metrics as FIRRTL passes. The generated instrumentation is independent of the backend simulator, showing how the concept of retargetable compiler also applies to RTL development. Finally, the formal verification infrastructure presented in Chapter 5 relies on a new backend to the FIRRTL compiler, which I developed to turn RTL designs into transition systems that a model checker can consume. The new `past` construct presented in that chapter is a good showcase of the extensibility of the FIRRTL compiler, which makes it possible to easily post-process circuit primitives through annotations that attach meta-data to them and automatically schedule associated compiler passes to be executed. The simplicity of the FIRRTL representation allowed these tools to work with virtually all Chisel circuits. In contrast, academic tools for SystemVerilog-based designs, for which no open-source compiler or IR exists, tend to be brittle and only work for benchmarks used in the respective papers.

Since Adam Izraelevitz graduated from UC Berkeley in 2019, most of the Chisel and FIRRTL infrastructure has transitioned from being maintained by graduate students to a team of full-time engineers primarily working for the SiFive semiconductor company. SiFive started re-implementing the FIRRTL compiler in 2020 under the leadership of Chris Lattner and using C++ and the MLIR compiler framework [82] to deal with larger designs and changing business requirements. Converting large designs at SiFive from FIRRTL to Verilog would often take several minutes up to half an hour with the old Scala-based research compiler. With the new compiler, this time has been drastically reduced [43].

Internally, SiFive builds most of its RISC-V CPU cores using Chisel. It then sells this IP to customers who receive the generated Verilog code. When these customers need to debug their design containing SiFive RISC-V cores, they thus need to rely on the output, rather than the input of the FIRRTL compiler, to investigate what is going wrong. This challenges the assumptions under which the original FIRRTL compiler was built. It is generally much easier, less error-prone, and compatible with many more tools to stick with simple Verilog constructs. Everyone who worked on the first Chisel-based designs at Berkeley had access to the original Chisel code. Thus, the generated Verilog could be treated as akin to assembly generated by a software compiler. In transitioning to the new compiler, SiFive engineers spent much energy and time to generate SystemVerilog output that would look better to customers by incorporating more advanced SystemVerilog features. While more bug-prone and time-consuming to implement, this approach does make sense when transitioning Chisel from academia to industry.

Open-source compilers and intermediate representations are not limited to hardware construction languages. There has been recent work in the area of high-level synthesis [112, 124, 129], as well as for established industry languages like Verilog [127, 153], and the CIRCT project which is trying to grow from a new implementation of the FIRRTL compiler to become a unifying compiler framework for hardware construction [43]. The open-source synthesis tool yosys also features an internal representation called RTL-IL [153]. Many ideas developed for this thesis can be applied to any of these new compilers, sometimes even for more significant gain. The CIRCT project, for example, is currently close to gaining a way to import SystemVerilog designs and compile them down to the simple core dialects. Thus, if someone re-implements the system for unified coverage that I represent in Chapter 3 for CIRCT, we would be able to obtain, e.g., a single toggle coverage pass that works not only for Chisel designs (via the FIRRTL frontend of CIRCT) but also for SystemVerilog and combined Chisel / SystemVerilog designs. Recently, I advised a Master's thesis on implementing some of the ideas I present in Chapter 5 with CIRCT, which will make them available for SystemVerilog designs once CIRCT gains the import capabilities [41].

## 2.5 Simulation-based Testing of RTL Designs

Designing hardware at the register transfer level can be difficult and error-prone, even with modern languages like Chisel. Digital hardware executes in a highly parallel manner; complex operations must be manually broken up and mapped to multiple cycles to meet timing; physical concerns like power efficiency must be addressed; and designers need to add extra complexity to enable testing the resulting semiconductor chip for manufacturing defects. Thus, it is paramount to test or verify each RTL hardware design comprehensively.

To obtain the highest level of fidelity and test execution speed, we would have to manufacture the hardware design. We would then be able to execute our tests on the final implementation. This approach is prohibitively expensive; circuit manufacturing takes a long time, and the visibility of the manufactured design is poor. Instead, the most common approach to executing an RTL description for testing is to use a software simulator or accelerated simulation with specialized chips. Testing designs in simulation is often referred to as dynamic verification in the hardware community.

### RTL Hardware Simulators

A variety of open-source and commercial simulators exist. Most of them focus on traditional hardware description languages like SystemVerilog and VHDL [136, 52]. Still, some recent options build on the FIRRTL compiler and thus exclusively work with circuits described in Chisel [97, 73, 10]. Software simulator implementations navigate a spectrum of latency vs. throughput regardless of the input language trade-offs. Simulators that interpret the RTL circuit description with little preprocessing are low latency in that they take very little time before simulation of the first cycle of execution. However, interpretation is generally slow; the number of cycles simulated per second, i.e., the throughput, tends to be low. Most commercial Verilog simulators started as interpreters because they are easy to implement [50] before switching to compiled simulation. The open-source simulator Icarus Verilog [151] and the Treadle [97] simulator for FIRRTL both use the interpreter approach.

The simulation throughput can be improved by converting the RTL circuit description into a compiled simulation. Open-source tools like Verilator [131] and ESSENT [10] achieve this by generating a simulation as C++ code, which is then compiled to optimized native machine code through a C++ compiler. Generating and compiling the C++ code takes significant time, thus increasing the latency but leading to much higher throughput.

For even higher throughput, we need specialized hardware that can take advantage of the massive amount of fine-grain parallelism contained in RTL designs [38]. RTL designs can be mapped to a network of small compute cores, enabling high throughput with some increase in latency that is required to perform the process of mapping the design across a large number of processing elements. Recent work has explored this option with custom bulk parallel processors as well as using off-the-shelf machine learning accelerators [45, 44]. There are also commercial products that use similar techniques, but details are hard to come by.

Even higher throughput can be achieved through FPGA-accelerated simulation as implemented by the FireSim tool [73, 76, 74, 94, 12] for Chisel designs. Unfortunately, mapping an RTL design to an FPGA after transforming it with the Golden Gate compiler [93] can take hours or nearly one day. In addition to that, the visibility is poor since the internal design state cannot easily be transferred out of a running design. There are options to gain more visibility. However, they require the design to be paused every time we need to inspect state [149]. Similar to the bulk synchronous option, there are also commercial products that use FPGAs to simulate RTL designs, but few details are publically available [12]. Simulating RTL designs on specialized hardware has the downside in that software testbenches can not easily be used to drive the design since the cost of communicating with the design on the simulation accelerator on a per-cycle basis is too high.

## Testbenches

The ability to simulate the execution of a given RTL hardware design is necessary but not sufficient to test the design. Most hardware designs are not closed systems. Instead, they feature input and output pins that communicate with the environment. In order to test a given design under test (DUT), we need to provide values to the input pins and check that the values on the outputs match our expectations. Often, we might also want to initialize states like registers and memories or check that they contain the values we expect them to contain. Thus, we need a setup called a testbench that lets us provide inputs to a DUT running in a simulator. This testbench generally takes on the form of a program interacting with the DUT, an event-driven model of some external hardware interacting with the DUT, extra RTL hardware used for testing that will not be included in the final design, or a combination of all three.

### SystemVerilog Testbenches

A simple testbench can be directly implemented in Verilog. Since Verilog is primarily a simulation language, it is not restricted to circuit descriptions but also features verification constructs. A Verilog testbench consists of a top-level module with no input or output pins - a closed system - which instantiates the DUT and connects to its input and output pins. The clock input of the DUT can be driven by toggling a variable between zero and one using a simple Verilog process and the delay construct to define the clock period. The other inputs are driven to chosen test inputs, also called test stimuli, through assignments in another process. Output values are checked with a simple if statement, and the finish statement can be used to terminate the simulation if there is a failure. Verilog also supports printing values to a file or standard output, which can be used to implement a check against a file containing the expected outputs. A recent study found that a large number of open-source RTL projects do not include self-checking testbenches [90]. Instead, a developer presumably checked the outputs as they were implementing the RTL design. These manual checks are often performed on a wave dump, i.e., a record of all signal values over time during a given simulation run. These wave dumps can be visualized with a waveform viewer like GTKWave or Surfer to aid debugging.

SystemVerilog [62] was developed to allow developers to program elaborate testbenches. It contains many features from general-purpose programming languages, like classes, methods, and functions, as well as specialized hardware testing features. SystemVerilog makes it possible to randomly initialize struct according to user constraints. This enables constrained random testing in which test engineers guide the random stimuli generation to explore the behavior of the design under test thoroughly. SystemVerilog Assertions provide an expressive, declarative temporal property language for runtime monitoring and formal verification. The Universal Verification Methodology (UVM) [63] is an industry-standard verification framework that takes advantage of many of SystemVerilog's new features. Unfortunately, no open-source simulator currently supports enough SystemVerilog features to run UVM testbenches. The reason for this is that a language that is meant to be useful for verification and hardware modeling incurs a lot of complexity, making it hard to implement support for the whole language.

### Simulator APIs

The open-source community and the vast majority of new hardware languages have thus taken a different approach. Instead of adding testing constructs to the hardware language, one can interface with the simulator from a general-purpose programming language and use that to implement a testbench. A classic example is Verilator, which converts an RTL design into the C++ source of a simulation of said design. The generated simulation code can then be instantiated and driven by a C++ program. As the testbench is compiled with the simulation, this approach leads to fast testbenches. Unfortunately, the API is fairly barebone, difficult to use, and simulator-specific, making testbenches non-portable.

### Testbench Libraries

Testbench libraries implemented in a general-purpose programming language provide a more powerful and user-friendly approach. Cocotb is an open-source library implemented in Python that allows users to interface with hardware designs simulated in a wider variety of commercial and open-source simulators [59]. Like Verilog, it uses an event-driven programming model, allowing users to implement processes that interact with the design under test based on Python async/await co-routines. Cocotb uses the standard DPI interface [62], which makes it compatible with many simulators. Unfortunately, DPI and the even-driven paradigm can lead to slow test execution. The Amaranth language features a testing library that uses Python co-routines for low overhead threading but interfaces directly with a Python-based simulator, thus reducing the communication overhead [150].

### ChiselTest

The ChiselTest library provides a rich Scala-based API to interact with Chisel-based designs under test [87, 42]. I was the maintainer and most frequent contributor between 2021 and 2024, taking over from Richard Lin, who started the project in 2018. ChiselTest originated from many prior attempts to support easy unit testing for Chisel designs and - in true Chisel tradition - eschews any event-based modeling. Instead, time only progresses when the clock of a synchronous design is stepped. Outputs that depend combinational on inputs take on their value immediately after an input has changed. This enables a simple interface, where users interact with a circuit by *poking* values to its inputs, *peeking* values at its outputs, and *stepping* the clock when needed.

This simple interaction model allows for more efficient simulator bindings. While Verilog simulators like VCS [136] and Icarus Verilog [151] are supported through DPI, similar to how cocotb works, ChiselTest generates a custom C++ harness for Verilator, which is then compiled to a shared library and loaded into the JVM before executing the Scala-based test. This architecture leads to fairly fast simulation speeds. For small designs, simulator startup time (latency) and the overhead of calling into a native simulator binary from the JVM prevail. Thus, ChiselTest also features a native FIRRTL interpreter called Treadle [97]. Since Treadle is written in Scala, the just-in-time compiler of the JVM can optimize the combination of test and simulator. Another option to avoid the overhead of calling into native code from the JVM is to turn a given circuit description into a simulation written in Java, similar to how Verilator generates C++ code. This approach was explored by two of my undergraduate advisees [47]. The Java code is compiled into byte code and loaded into the JVM, enabling even better cross-optimizations between testbench and circuit simulation. However, the startup overhead is higher than that of Treadle due to the compilation steps involved.

We have previously discussed how hardware designs are highly parallel. Thus, it is no surprise that more sophisticated testbenches might also benefit from concurrency. Often, multiple interfaces on the DUT need to be exercised, which is most naturally expressed as multiple threads of execution. These threads generally yield for a single clock event. Thus, it is often the case that each testbench thread needs to be scheduled at least once for every simulated clock cycle. Some designs simulate at multiple MHz, thus requiring several million thread switches per second. This is the reason why Python-based frameworks all use co-routine-based threads. Unfortunately, such lightweight, compiler-based concurrency techniques did not exist on the JVM until recently. Thus, ChiselTest is forced to use JVM threads, which generally map onto relatively heavy-weight operating system threads. This leads to relatively large slowdowns for multi-threaded testbenches. This effect is most prominent for small designs that simulate quickly; larger designs will spend more simulation time per cycle, and thus, the overhead is less noticeable.

### Staged Test Execution

So far, all hardware testing libraries we discussed are based on an eager evaluation of simulation constructs. Inputs and outputs are immediately updated, and new values are retrieved from the simulator. This way, testbenches can easily use facilities of the host language to generate inputs on the fly and to process outputs with complicated checkers. It also allows users to easily single-step through their testbenches using a standard Python or Scala debugger. On the other hand, the *fault* testing library implements its testing API as a deeply embedded domain-specific language [143]. Instead of directly communicating with the simulator, a testbench program is generated by executing a Python program, similar to a Python-embedded HCL, which would generate a hardware description. The syntax tree of the testbench program can then be lowered to a much wider variety of testing backends. Beyond simple RTL hardware simulators, *fault* also targets analog circuit simulators and formal verification tools. The extended flexibility comes with a much higher implementation complexity since *fault* cannot take advantage of Python constructs for control flow or computation. Instead, everything the test should do must be expressable in the *fault* IR.

## Coverage

Since dynamic verification looks at concrete executions of the design, a feedback mechanism is needed to know whether the simulated executions explore all interesting behaviors of the DUT. To this end, various notions of coverage have been developed [117]. While high coverage is insufficient to declare the verification process a success, it can serve as a progress indicator and help test engineers decide which parts of the DUT to prioritize when writing new testbenches. Coverage metrics broadly fall into two categories: structural or automated coverage and functional coverage.

When an RTL designer writes a line of hardware description code, we assume it is there for a reason. Without that line, the hardware would malfunction. This simple intuition forms the basis of all structural coverage metrics. Since these metrics only need to analyze the structure of the hardware description, they generally do not require additional user input. They can be implemented fully automatically by a given hardware simulator. We have already introduced the concept of line coverage, where we are trying to see if all lines of a given hardware description code have been executed. Another popular automated metric is toggle coverage. This coverage metric checks for every signal in the circuit how many bits are toggled from one to zero or from zero to one at least once. If a signal bit always carries the same constant value during all tests, it could either be removed or there is a behavior not covered by the current set of testbenches. Finite state machine (FSM) coverage is a structural metric that can be automated if state machines are explicitly encoded in the target language. For SystemVerlog and Chisel, this is not the case. However, there are only a handful of recommended patterns, such that – in many cases – FSMs states and transitions can be inferred, and thus coverage can be collected automatically. Chapter 3 details how I implemented line, toggle, and FSM coverage for Chisel-based designs in a manner that works with a wide variety of available simulators.

While structural coverage is very easy for RTL designers to use – they just need to enable the respective feature in their simulator – it does lack high-level design insights and can thus be misleading. It might also generate some noise in that missing toggle coverage might indicate a bug, but it could also often just not be worth achieving since no additional high-level functionality is tested by trying to get to full toggle coverage. This problem is solved by functional coverage. Instead of relying on the circuit simulator to automatically infer low-level functionality from the RTL description, functional coverage requires RTL designers to annotate high-level design functionality and intent manually. For example, the designer might annotate the combination of signals that indicate that a processor pipeline resolves a read-after-write hazard. This functional coverage point can track how often each testbench exercises this feature. While defining good functional coverage can be quite a challenge for the RTL designer, it is quite easy to implement from the simulator's perspective since it is completely user-defined and does not require RTL analysis. However, there is an interesting design space of frontend language features that simplify the process of defining functional coverage points. SystemVerilog, for example, features a rich vocabulary of high-level coverage definitions. In Chisel, a similar result could be achieved by a generator library for coverage. However, no such library has been implemented so far.

## 2.6 Formal Verification of RTL

Simulator-based dynamic verification means executing the DUT concretely using a mix of manually defined or randomly generated test inputs. Unfortunately, the space of possible inputs for many designs is extremely large ($2^{inputbits}$) for even a single execution cycle. It is essentially infinite if we want to guarantee that the design will execute correctly, no matter how many cycles it is executed for. This does not mean that dynamic verification is ineffective at finding bugs. Still, for most realistic designs, we won't be able to guarantee that no bugs exist and need to use coverage as a crutch to ensure that we tested enough diverse inputs.

An alternative is formal verification, the umbrella term for various techniques that aim to analyze the design to prove correct functionality mathematically. In the hardware domain, these proofs are generally computer-assisted to scale to realistic designs. Properties can be proven with interactive proof assistants [30, 29, 46, 146] or more automated methods like a combination of symbolic execution and abstract interpretation [57, 155] or model-checking based on binary decision diagrams (BDDs) [23, 99], satisfiability (SAT) [98] or SMT [9, 8]. In this thesis, we focus on SAT and SMT-based methods.

The boolean satisfiability problem poses the following question: Given a boolean formula, does there exist an assignment to the inputs such that the formula evaluates to true? This problem was proven to be NP-complete and thus might take a prohibitively long time to solve for general problem instances. However, researchers have found heuristics that work well on a large set of SAT instances that reflect real-world problems. Nowadays so, so-called SAT solvers can solve formulas with many input variables [98]. However, there will always be even small instances that cannot be solved within a reasonable amount of time; it just so happens that these are often not reflective of real-world problems.

As discussed in Section 2.1, any RTL circuit can be described by boolean functions that model how outputs and state updates are calculated. Thus, if we are provided with an assertion, i.e., a boolean formula over the inputs and states in the circuit that we expect to always hold, we can ask the question of whether there exists an assignment to the circuit inputs and the circuit state such that the assertion is violated. We can expand our check to more than a single cycle of execution by *unrolling* the RTL design. This means that we combine output and state update functions to describe the assertion truth-value as a boolean function over the starting state as well as the inputs in steps one and two. This technique is called bounded model checking [13] and can be implemented with an RTL compiler that takes a given circuit and compiles it down to a boolean formula in conjugating normal form (CNF), which can then be provided to a SAT solver. If the solver provides an assignment, we have found a way to drive the design into a state where it violates our assertion. Otherwise, we are guaranteed (assuming the SAT solver and our conversion do not have bugs) that the assertion holds for at least $k$ cycles where $k$ is the number of unrollings.

Instead of lowering an RTL design directly into a boolean formula, modern *word-level* model-checking implementations generate a first-order logic formula using the theory of bit-vectors and arrays. These theories are standardized as part of the SMTLib specification [8]. The process of generating the formula is essentially the same as turning behavioral into structural RTL as described in Section 2.1. A large number of different SMT solvers are able to process these formulas and check them for satisfiability. The higher level of abstraction allows SMT-solvers to perform word-level rewrites, often eliminating large parts of a formula before bit-blasting it [109]. By using the theory of arrays, we avoid having to model each memory bit of every memory entry as a boolean variable, as would be required for bit-blasting. SMT-based model checking is implemented by a range of tools like AVR [54], Pono [96] and SymbiYosys [152]. Chapter 5 details my implementation of an SMT-based model checker for Chisel, released as part of the ChiselTest library.

Bounded model checking is beneficial for finding bugs since it will be able - at least in theory - to provide us with a concrete testbench or waveform that we can use to debug the counter examples as if a concrete simulation test discovered it. However, if no inputs that violate an assertion are discovered, BMC only guarantees that no bug can be found if the design is executed for up to $k$ cycles. In practice, BMC often becomes exponentially slower with increasing $k$, and thus, realistic values for $k$ are often under 100 steps, while the final circuit will execute at millions or billions of cycles per second. Model-checking with BDDs can provide unbounded guarantees because it is able to detect when the BDD has reached a steady state, and thus all reachable states are described by it. However, model checking with BDDs generally does not scale well to larger circuits. SAT and SMT-based model checking can deal with larger designs, and several approaches to unbounded proofs have been developed. The simplest is k-induction [130], which tries to prove that the assertion holds through induction over $k$ unrollings. K-induction often requires engineers to manually supply strengthening invariants to restrict the state space. Property-directed reachability

(PDR, also called IC3) automatically generates invariants but is much harder to debug when it fails or times out. Neither k-induction nor PDR can provide the user with a true counter-example demonstrating the bug. If no proof is found, this does not necessarily mean that the assertion is falsifiable; instead, better-strengthening invariants might be needed. Deciding whether a failing proof is caused by an actual bug is left to the verification engineer. Often, BMC is employed in conjunction with proof techniques to rule out any bugs that can be easily discovered with BMC.

# Chapter 3

# Simulator Independent Coverage

Simulation-based testing, as introduced in Section 2.5, is the most commonly used technique to verify RTL designs. Simulators, waveform viewers, and testbench libraries are readily available and easy to use. However, to assess the quality of our tests, we require feedback on how well they exercise the functionality of the design under test. While the coverage metrics that we introduced in Section 2.5 are well known to provide valuable feedback to test engineers, they are not well supported among new hardware construction languages and simulators. Instead of following the path taken by the old generation of commercial simulator tools for Verilog, we propose a new approach to coverage collection that leverages the advantages of the modern open-source eco-system for hardware languages. Our approach was published at ASPLOS in 2023 [80]. We identified several issues with the status quo of coverage instrumentation and collection that need to be addressed:

1. Most open-source or innovative research simulators lack support for collecting and reporting automated coverage metrics [10, 97, 131, 73].

2. For tools that do support these metrics, their custom implementation makes merging coverage across various software or FPGA-accelerated simulators and formal tools difficult.

3. New hardware languages generally lack support for source-level coverage metrics. While we can get coverage metrics for the generated Verilog, there is no automated way to map the coverage results back to the original Chisel code.

In this chapter, we present our new approach that relies on a compiler to lower common automated coverage metrics to a single `cover` primitive that can be easily implemented for a wide range of different simulators. Each metric is implemented as a compiler pass that generates only `cover` primitives in addition to synthesizable constructs, which are already supported by all simulators. It also collects metadata that allows a report generator to map the coverage counts back to the high-level information, such as which lines were covered. The simulator implements the `cover` primitive as a counter, which is incremented every time the input signal is true at a clock event and reports back the counts at the end of the simulation. A simulator-independent report generator then consumes the metadata from the compiler pass as well as the cover counts from the simulator and thus creates a user-readable report. Since the coverage counts reported by simulators are all in the same format, we can trivially merge results from different simulators before extracting the high-level coverage reports. Figure 3.1 provides an overview of our system.

We implemented our approach for the Chisel hardware construction language and the FIRRTL compiler [7, 67]. Over a short period of time, we were able to implement line, toggle, and finite state machine coverage, thus exceeding the number of automated coverage metrics offered by any open-source RTL simulator today. For the coverage metrics that are natively supported by the open-source Verilator simulator, we found no slowdown for our simulator-independent solution. While implementing new coverage metrics can be challenging, adding support for new simulators was fairly simple. Besides Verilator, we also provide support for a FIRRTL simulator called treadle, for the ESSENT simulator [10], the FPGA-accelerated FireSim simulator [73] as well as a formal tool for trace generation.

### Automated Coverage on the Structural Verilog

Since new hardware construction and high-level synthesis (HLS) languages generate structural Verilog in order to target existing backend tools, one may think that the easiest way to get automated coverage would be to just use the coverage collection flags that are already built into the existing Verilog simulators. However, automated coverage generally relies on patterns in the code that are written by the designer.

For example, if we create a mux with a branch statement (`if` in Verilog, `when` in Chisel), the condition will be taken into account when calculating line or branch coverage. However, if we create a mux through a conditional assignment or through an explicit exclusive-or gate, it does not show up in the line coverage report. Figure 3.3 shows an example where a branch in Chisel gets lowered into a conditional assignment by the FIRRTL compiler in order to simplify the structural Verilog generation. This is perfectly valid, as it preserves the semantics of the original Chisel code [1], however, it means that achieving 100% line coverage on the generated Verilog may not always result in complete line coverage for the original code written by the designer.

Another example is finite state machine (FSM) coverage: While the pattern that designers use for FSMs is clear in the original Chisel, it is not recognized by Verilog simulators that only have access to the generated structural Verilog.

## 3.1 Simulator Independent Coverage Interface

A typical Chisel testing flow can involve different simulators, depending on the desired start-up speed, throughput, and debugging features. In order to support coverage on all of them,

[1]In Verilog, changing a branch into a conditional assignment changes the semantics of the code due to X-propagation. In Chisel, there is no X-propagation, and thus the semantics are preserved.

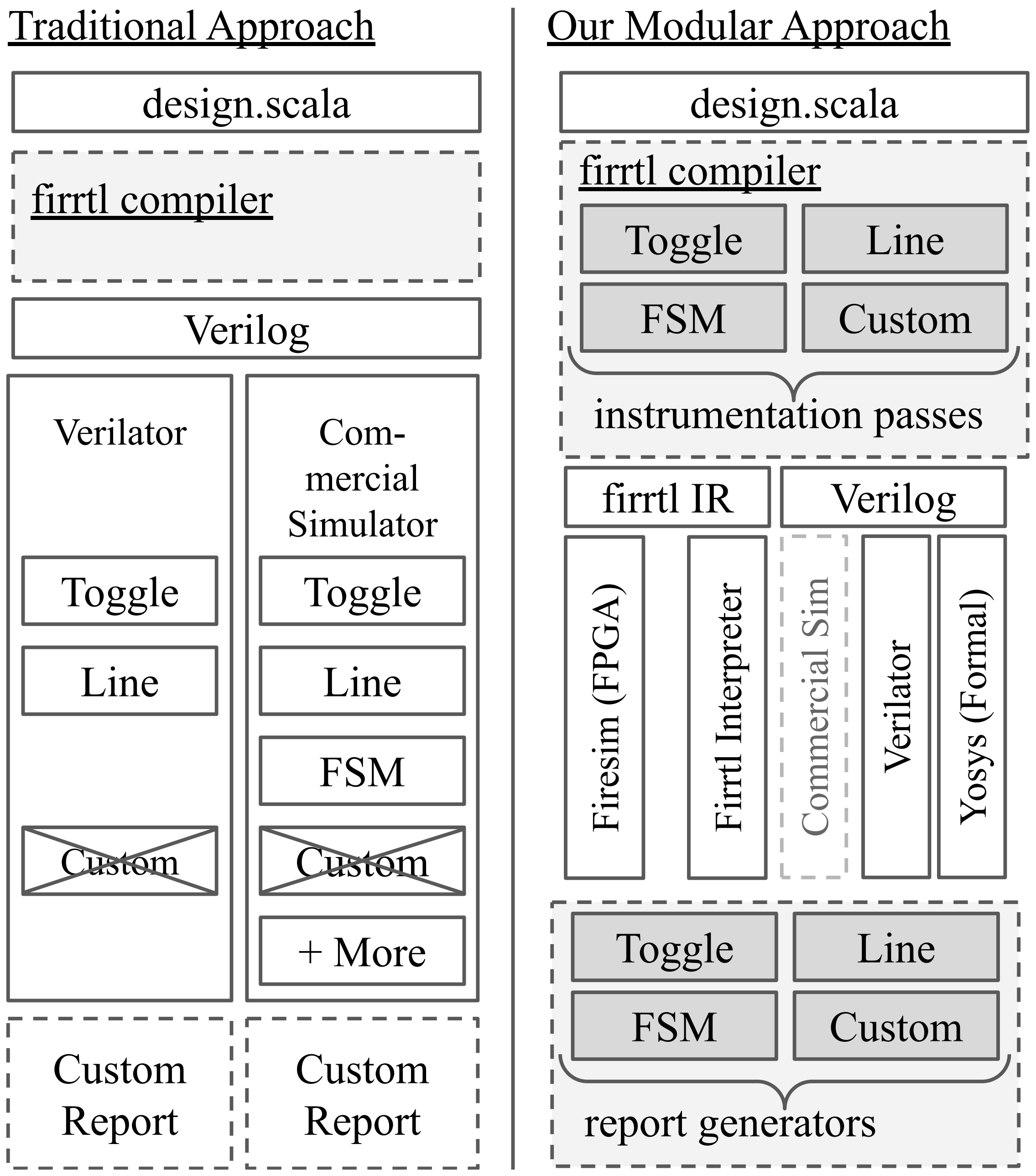
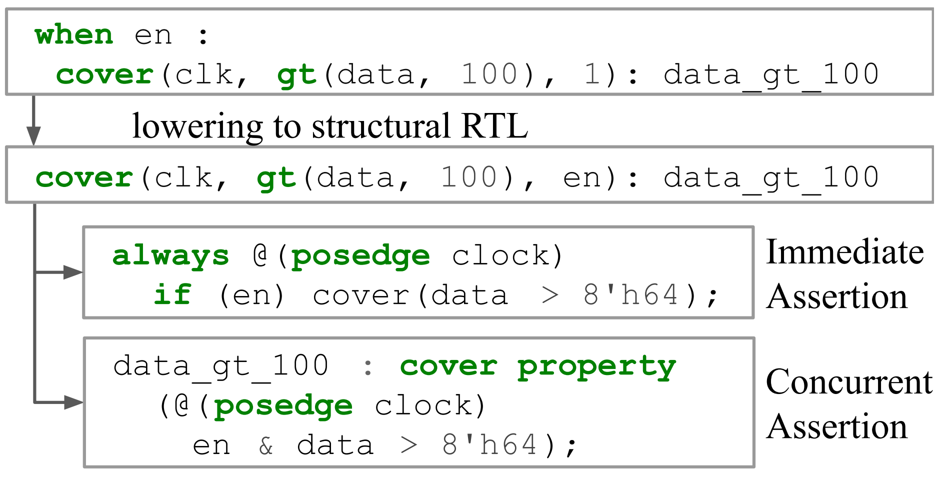


Figure 3.1: **Overview:** Traditionally, automated coverage collection is part of a monolithic simulator. Users are limited to the coverage metrics that the simulator authors have chosen to provide. We instead implement every coverage metric as a single instrumentation pass in the FIRRTL compiler and a simulator-independent report generator. Only support for our proposed `cover` primitive needs to be added to a new simulator to take advantage of all our coverage metrics.



```
when en :
 cover(clk, gt(data, 100), 1): data_gt_100
```

```
cover(clk, gt(data, 100), en): data_gt_100
```

```
always @(posedge clock)
  if (en) cover(data > 8'h64);
```

```
data_gt_100 : cover property
  (@(posedge clock)
    en & data > 8'h64);
```

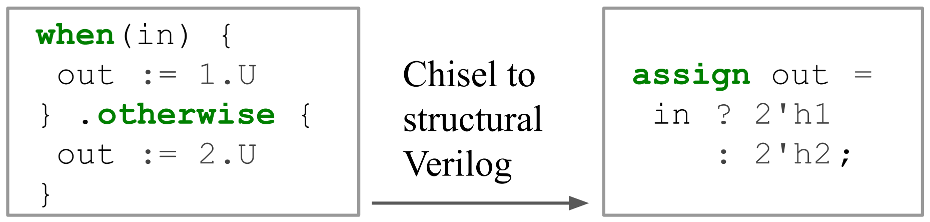
Figure 3.2: After lowering the `cover` statement to structural RTL, it can be emitted to SystemVerilog as an immediate or concurrent assertion.

we developed a simple interface that takes advantage of existing coverage features in Verilog simulators and can easily be implemented for the five very different verification tools that we worked with.

All our simulators support simulating any synchronous RTL circuit that can be generated from Chisel. The one IR primitive we add is a `cover` statement which samples a signal on the rising edge of a clock and increments a counter if and only if the covered signal is true. Each cover statement also carries a name that uniquely identifies it inside the module it is declared in. This way, simulators can report coverage results as a simple map from the cover statement's name (including its path in the module instance hierarchy) to a non-negative integer representing the count. Different simulators may use counters with different bit-widths as long as the count is saturating. This allows important optimizations in FPGA-accelerated simulators. We implemented support for the `cover` statement in five different backends.

## Treadle

Treadle [97] is a Java Virtual Machine based simulator for circuits represented in the FIRRTL IR. While it does not achieve the simulation speeds possible with a compilation-based approach, it features quick spin-up times and integrates well into the Scala-based Chisel ecosystem, and is thus the preferred simulator for shorter simulation runs and smaller- to medium-sized designs. Adding support for the `cover` statement took less than one work week and around 200 lines of Scala code. Treadle had existing support for a `stop` statement which also samples a condition at a positive edge. This code was easy to adapt for the `cover` statement – we just needed to increment a counter when the condition is true instead



```
when(in) {
 out := 1.U
} .otherwise {
 out := 2.U
}
```

```
assign out =
 in ? 2'h1
    : 2'h2;
```

Figure 3.3: In this example, the translation to structural Verilog replaces a branch with a conditional assignment. Therefore, 100% line coverage on the generated Verilog does not necessarily imply complete line coverage of the Chisel source.

of stopping the simulation. At the end of the simulation run, all counts are transferred into a map from a cover point name to a count.

## Verilator

Verilator [131] is a popular open-source Verilog simulator. It analyzes and optimizes the input Verilog and generates C++ source code for a simulation which is then compiled to a binary with a standard C++ compiler. This approach generally leads to higher simulation speeds, but it does increase the time spent building the simulation, which is why it lends itself to longer simulation runs where the startup cost can be amortized.

In order to simulate a Chisel design, it needs to be compiled into structural Verilog which will then be turned into a simulation by Verilator. Our `cover` statement can be mapped to a concurrent or an immediate assertion in the Verilog generated by the FIRRTL compiler as shown in Figure 3.2. By default, we generate immediate cover statements [62] as those are the only form supported by the open-source Yosys [153] synthesis tool covered in Section 3.1.

This way we make use of the built-in support for user-defined coverage in Verilator. We do not re-use any of the Verilog line- or toggle coverage provided by Verilator. At the end of the simulation run, Verilator generates a coverage data file that contains the counts associated with each SystemVerilog cover statement. We also implemented a converter that parses the custom coverage format used by Verilator and re-associates the counts with the `cover` statements in the FIRRTL source. Our interface code thus generates the exact same map from `cover` statement names to counts as provided by our native implementation for Treadle.

## FireSim

FireSim [73] is an open-source, cycle-accurate FPGA-accelerated RTL simulator, which at its core, uses a custom compiler based on FIRRTL [93] to decouple the clock of the simulated RTL from the FPGA clock. This allows for deterministic, cycle-accurate composition with software simulations of components like network switches and FPGA-optimized multi-cycle

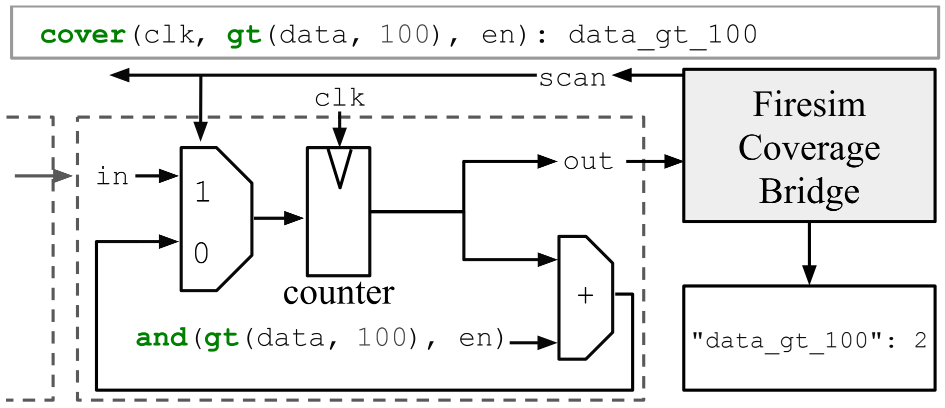


Figure 3.4: We generate saturating counters and a scan chain for all `cover` statements for FPGA-accelerated simulation with FireSim.

simulation models, e.g., of multi-ported register files or DRAM models with realistic access latencies.

While FireSim's compiler supports some conventionally non-synthesizable debug primitives, like assertions and prints, it currently has no means to implement `cover` statements, which cannot directly be mapped onto an FPGA. We added a new compiler pass to FireSim which replaces each `cover` statement with a saturating counter that is then connected to a per-clock-domain scan chain (Figure 3.4). The counter's bit width is a parameter set by the user to trade off FPGA resources and cover count accuracy. The pass also generates a list with the names of all `cover` statements in the order in which they are connected throughout the scan chain. The scan chain is controlled by an FPGA-hosted simulation module and C++ driver program, which can pause the simulation, freeze all coverage counts, and then clock out all coverage counts. Using the metadata generated by the newly added coverage scan-chain insertion pass for FireSim, we can then map the counts to the `cover` statement names. We thus get the exact same coverage information from the FPGA-accelerated simulation as provided by the software simulators Treadle and Verilator.

## Formal Verification with SymbiYosys

The most common use of formal verification tools is to verify assertions. The tool will either find a series of inputs that lead to an assertion violation or provide a proof showing that the assertion can never be violated. In addition to that, the open-source SymbiYosys tool (like many commercial tools) also supports coverage trace generation [152]. Given a design annotated with cover points, it will try to find sequences of inputs that will lead to each of the cover points. Since we already emit our `cover` primitive as a standard immediate assertion for the Verilator simulator, the same generated Verilog can be used by SymbiYosys to automatically find inputs that will maximize any of our automated coverage metrics.

Table 3.1: Lines of code (LoC) for coverage passes and report generators. Lines of new library code in parenthesis.

| | LoC Instrum. | LoC Report |
|---:|---:|---:|
| Common Library | 106 | 290 |
| Line Coverage | 89 | 64 |
| Toggle Coverage | 279 (+131) | 51+ |
| FSM Coverage | 144 (+228) | 34 |
| Ready/Valid Coverage | 78 | 26 |

### ESSENT

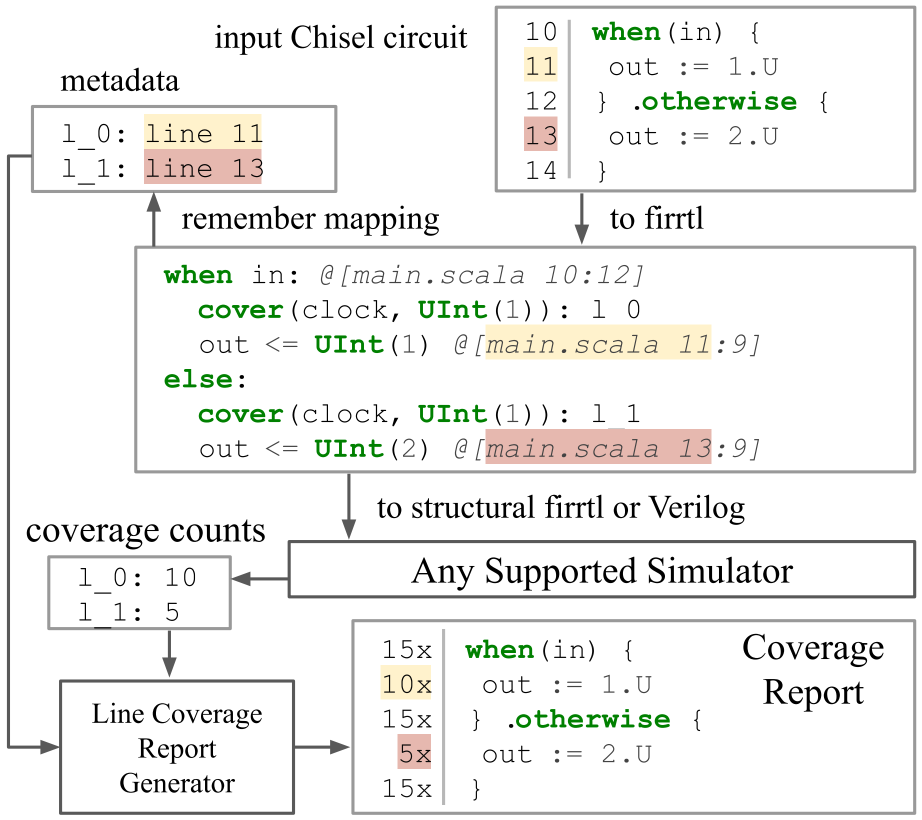
ESSENT is a high-performance simulator prototype [10] with no debugging support. After the basic idea had been validated with the other four backends, we recorded the time spent to get a sense of how hard it would be to add support for a fifth tool. Overall, it took us around 5 hours and 60 lines of code to add support for our `cover` primitive and thus allow ESSENT users to make use of all our coverage metrics.

## 3.2 Coverage Instrumentation and Report Generators

In this section, we describe how we implemented a number of automated coverage metrics. Our methodology relies on the assumption that most automated coverage metrics can be implemented using the `cover` statement introduced in Section 3.1. To demonstrate this, we implemented line coverage, toggle coverage, and FSM coverage as well as a custom Ready/Valid coverage metric. Each metric is implemented as an instrumentation pass that analyzes the circuit represented in the FIRRTL IR, adds cover statements and emits metadata as well as a report generator that consumes the simulator output and turns it into a high-level coverage report. To provide a sense of implementation complexity, Table 3.1 contains an overview of all coverage metrics implemented for this chapter, along with the number of lines of Scala code for the associated instrumentation and report generator. All our report generators are bare-bones and generate simple ASCII reports only. Many potential improvements could be made to generate interactive HTML reports or similar, which would significantly increase the amount of code in the report generators.

### Branch and Line Coverage

Branch coverage counts how often a branch is taken in the HDL or HCL source code. Line and statement coverage can be derived from this information by counting the number of lines or statements that are executed when a particular branch is taken. In order to implement a branch coverage instrumentation pass we rely on the fact that the FIRRTL compiler



```
10  when(in) {
11    out := 1.U
12  } .otherwise {
13    out := 2.U
14  }
```

```
l_0: line 11
l_1: line 13
```

```
when in: @[main.scala 10:12]
  cover(clock, UInt(1)): l_0
  out <= UInt(1) @[main.scala 11:9]
else:
  cover(clock, UInt(1)): l_1
  out <= UInt(2) @[main.scala 13:9]
```

```
l_0: 10
l_1: 5
```

```
15x  when(in) {
10x    out := 1.U
15x  } .otherwise {
 5x    out := 2.U
15x  }
```

Figure 3.5: The line coverage pass instruments every `when` statement in the FIRRTL circuit. As part of the standard lowering in the FIRRTL compiler, the branch conditions will be pulled into the enable condition of the cover statement. The mapping from lines to branches is used to generate the coverage report from the counts reported by the simulator.

automatically turns the dominating branch condition of a statement into an enable signal for the statement. This is done during lowering to structural RTL as shown in Figure 3.2. Thus, we place our instrumentation pass before that lowering happens and just add a `cover` statement right after every branch. This is shown in Figure 3.5. We also make sure to skip branches that have no statements in them or that only contain a `cover` statement inserted by the designer, as trying to cover these branches would not be helpful.

In order to turn the branch coverage information into actual line coverage, additional information is needed. We scan all statements directly inside a given branch and extract their line numbers and source file information. Thus, we build up a map from a cover point to the lines it covers. After the simulation finishes, the map is used by our report generator to turn coverage counts from the simulator into a textual report that annotates the Scala source file with counts of how often each line was executed.

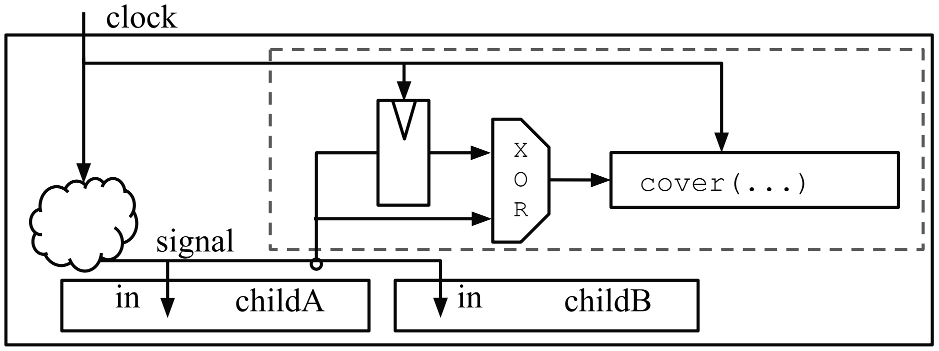


Figure 3.6: The toggle coverage pass adds a register and a xor gate. It avoids redundant instrumentation for signals that always have the same value.

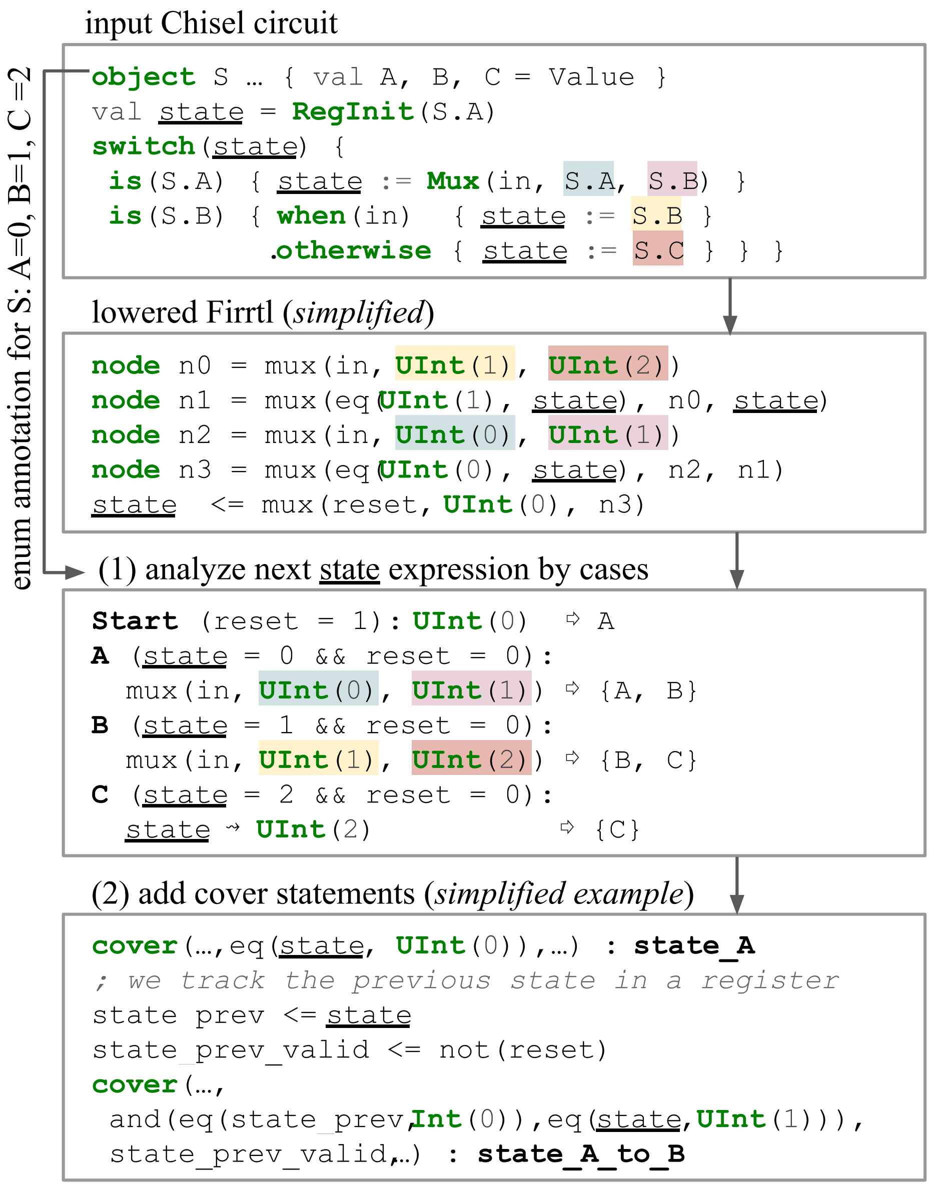
## Toggle Coverage

We implemented our toggle coverage as a compiler pass that runs on the structural RTL after optimizations such as constant propagation and dead code elimination have been performed. We distinguish between I/O signals, registers, memories, and wires and allow the user to choose which category they want to instrument. For every selected signal, we add a register in order to record its value in the previous clock cycle. A xor gate allows us to detect whether a bit in the signal changed. Counting rising and falling edges, i.e., toggles from zero to one or one to zero, separately would be a simple extension that would use two instead of one `cover` statement per bit. We also add a register that is zero in the first cycle of the simulation and one after in order to disable all toggle `cover` statements during the first cycle when the previous value has not been updated yet.

We implemented a global alias analysis, which analyzes the design hierarchy and reports groups of signals that are guaranteed to always carry the same value. For example, in Figure 3.6 the "signal" wire in the top module always carries the same value as the "in" ports of the two child modules. Our toggle coverage pass uses the alias information to instrument only a single signal from each alias group. An important example is the global reset signal, which is instrumented only once in the top-level module instead of once in every module in the hierarchy. The global alias analysis pass is necessary to make toggle coverage perform well.

## Finite State Machine Coverage

Finite State Machines (FSMs) are commonly used to implement the controls for RTL modules. In modern Chisel, designers generally create a `ChiselEnum` that contains all the states and a state register of their custom enum type. To implement our instrumentation pass, we take full advantage of the annotation system which allows Chisel libraries, like the `ChiselEnum` library, to annotate circuit elements in Scala. We use the annotation to find


input Chisel circuit
lowered Firrtl (simplified)
(1) analyze next state expression by cases
(2) add cover statements (simplified example)
enum annotation for S: A=0, B=1, C =2

```
object S … { val A, B, C = Value }
val state = RegInit(S.A)
switch(state) {
 is(S.A) { state := Mux(in, S.A, S.B) }
 is(S.B) { when(in)   { state := S.B }
          .otherwise { state := S.C } } }
```

```
node n0 = mux(in, UInt(1), UInt(2))
node n1 = mux(eq(UInt(1), state), n0, state)
node n2 = mux(in, UInt(0), UInt(1))
node n3 = mux(eq(UInt(0), state), n2, n1)
state  <= mux(reset, UInt(0), n3)
```

```
Start (reset = 1): UInt(0)  ⇨ A
A (state = 0 && reset = 0):
  mux(in, UInt(0), UInt(1)) ⇨ {A, B}
B (state = 1 && reset = 0):
  mux(in, UInt(1), UInt(2)) ⇨ {B, C}
C (state = 2 && reset = 0):
  state ⇝ UInt(2)           ⇨ {C}
```

```
cover(…,eq(state, UInt(0)),…) : state_A
; we track the previous state in a register
state_prev <= state
state_prev_valid <= not(reset)
cover(…,
 and(eq(state_prev,Int(0)),eq(state,UInt(1))),
 state_prev_valid,…) : state_A_to_B
```

Figure 3.7: Finite state machine (FSM) coverage assumes that the state register uses a `ChiselEnum`. We first analyze all possible next states by simplifying the state update expression for each possible current state. We then add `cover` statements for all states and possible transitions.

Table 3.2: Software simulation benchmarks and the number `cover` points generated by the line and toggle coverage instrumentation passes.

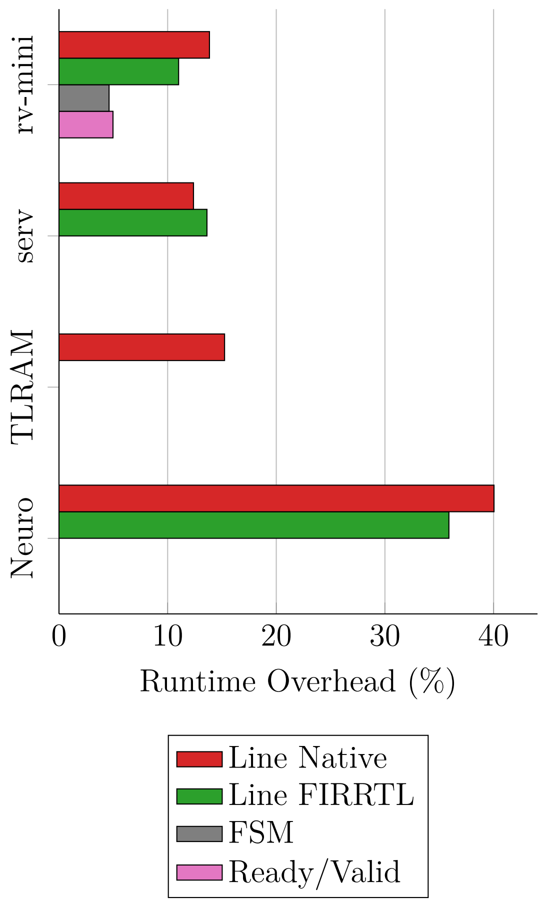
| Design | Cycles Executed | Run Time | # Line | # Toggle |
|---:|---:|---:|---:|---:|
| riscv-mini [75] | 126,550 | 3.34 s | 157 | 4,042 |
| TLRAM [6] | 816,473 | 1.45 s | 8 | 2,532 |
| serv-chisel | 828,931 | 1.05 s | 79 | 725 |
| NeuroProc [121] | 53,455,204 | 40.38 s | 809 | 4,786 |

registers that contain values from a `ChiselEnum`. The annotation also tells us all legal states that were defined as part of the enum. Figure 3.7 shows an example where the enum `S` contains the three possible values `A`, `B`, and `C`.

With this information, we analyze the next expression of the register. In our example, there are four cases that we need to analyze: One case when the system is in reset, and one case for each possible state. In each case, we apply constant propagation, replacing the `reset` and `state` symbols with their assignments. Thus, we collect all possible next-state assignments and derive all possible transitions. In cases where – after simplification – we end up with an expression that is not a constant or a mux, we over-approximate, assuming all states are possible next states. Thus, our analysis is conservative in that it will only over-report possible transitions and never miss any transitions. One example where our analysis fails is an FSM in RocketChip [6] where the next state signal goes through a submodule that is invisible to our (module scoped) analysis. After analyzing the possible transitions, we add cover points for every state and transition in a second step.

## Ready/Valid Coverage

One of the most commonly used interfaces from the Chisel standard library is a `DecoupledIO` bundle. A data transfer happens during cycles in which the `ready` and `valid` wires are both asserted. We developed a custom coverage pass that analyzes the ports of all modules in the design and adds a `cover` statement for every decoupled interface it finds in order to count how often data is transferred. Thanks to all the code we had previously developed, we were able to implement and test this new coverage metric in around 3h. This shows how new metrics that may be specific to a design ecosystem can easily be added by using our simulator-independent approach. Traditionally RTL designers might have manually added cover statements as part of the functional coverage, however, our pass is more economical since it works across a wide range of designs using `DecoupledIO` without manual annotations.



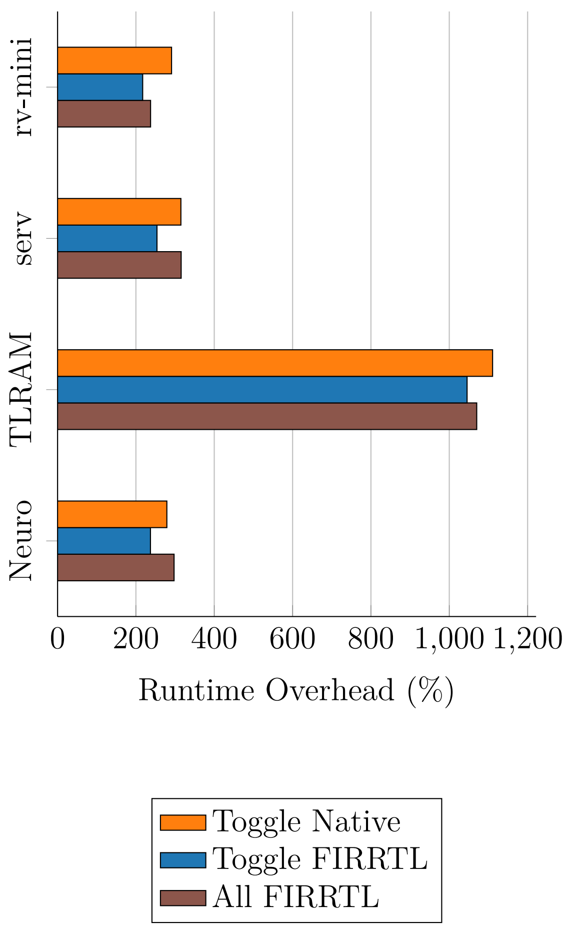


Figure 3.8: Coverage instrumentation overhead with Verilator v4.034. For TLRAM, the measured overhead of our FIRRTL line coverage is close to zero.

## 3.3 Evaluation

Prior sections already discussed how our approach allowed us to quickly implement four different coverage metrics for five different backends. In this section, we investigate the run time and/or area overhead of our simulator-independent coverage solution.

### Software Simulator Coverage Overhead

Making use of generic `cover` statements instead of hard-coding line coverage into the simulator allows us to support new simulators with little effort. However, one might suspect that using a more generic coverage collection mechanism compared to built-in line coverage might create additional overheads. We measure the overhead of our coverage metrics on simulation speed and compare it to the built-in Verilog coverage of the open-source Verilator simulator.

Our benchmarks come from various open-source projects written in Chisel. Table 3.2 provides an overview. We picked long-running tests, recorded a waveform VCD and then generated a minimal testbench that only replays the top-level inputs from the VCD. This way we can isolate the simulator run time from the time it takes to generate stimuli and any overhead in the verification environment. This careful isolation means that the reported overhead may be less noticeable in practice [2].

Figure 3.8 shows the run time overhead of various coverage instrumentation over the baseline. We find that in general, our instrumentation causes the same or slightly less overhead compared to Verilator's built-in coverage. This can be attributed to the fact that Verilator appears to internally follow an approach similar to ours.

While we are prohibited from reporting data for commercial simulators in a meaningful way, we observed that our generic approach does negatively impact the performance of event-driven simulators. However, Verilator with our coverage is generally significantly faster than any commercial tool with its native coverage. By providing extensive coverage support for open-source simulators, we remove one of the common reasons that prevent users from switching to faster simulators like Verilator.

## FireSim Coverage Overhead

We applied our line coverage instrumentation to two different SoC designs from the Chipyard framework [5]. The first includes four, in-order scalar Rocket [6] cores and the second uses a single out-of-order BOOM [160] core. This results in 8060 `cover` statements in the RocketChip design and 12059 `cover` statements for the BOOM SoC. We then ran our scan chain insertion pass and transformed the designs into a cycle-accurate FPGA-accelerated simulation with FireSim [73]. Both simulators target a Xilinx Ultrascale+ VU9P device, the FPGA supplied by Amazon EC2 F1 instances, and were compiled using Xilinx Vivado 2018.3.

Figure 3.9 shows the resource usage for different counter sizes and compares them to a baseline without any coverage instrumentation. We include numbers for up to 48 bit of resolution which would be sufficient to prevent counter-saturation in practically all applications. Wide coverage counters lead to significant increases in resource usage, but as long as we are only interested in finding lines that have never been covered, small counters offer minimal area overhead. Figure 3.10 illustrates the $f_{max}$ scaling trends versus increasing counter widths. For counter widths up to 8 bit for the Rocket and 2 bit for the BOOM-based design, the overhead from our coverage support falls within the noise introduced by differing placements.

We used our instrumented SoCs with 16 bit coverage counters to boot Linux and obtain line coverage results. For the RocketChip design the simulation executed 3.3 B cycles in 50.4 s

[2]Chisel testbenches normally slow down the simulation by 2x-1000x. Industry insiders tell us that well-optimized commercial SystemVerilog testbenches often present a 50% overhead leading to a 2x slowdown compared to raw simulation speed. Thus the raw simulation overhead that we measured will be less noticeable with a real testbench.

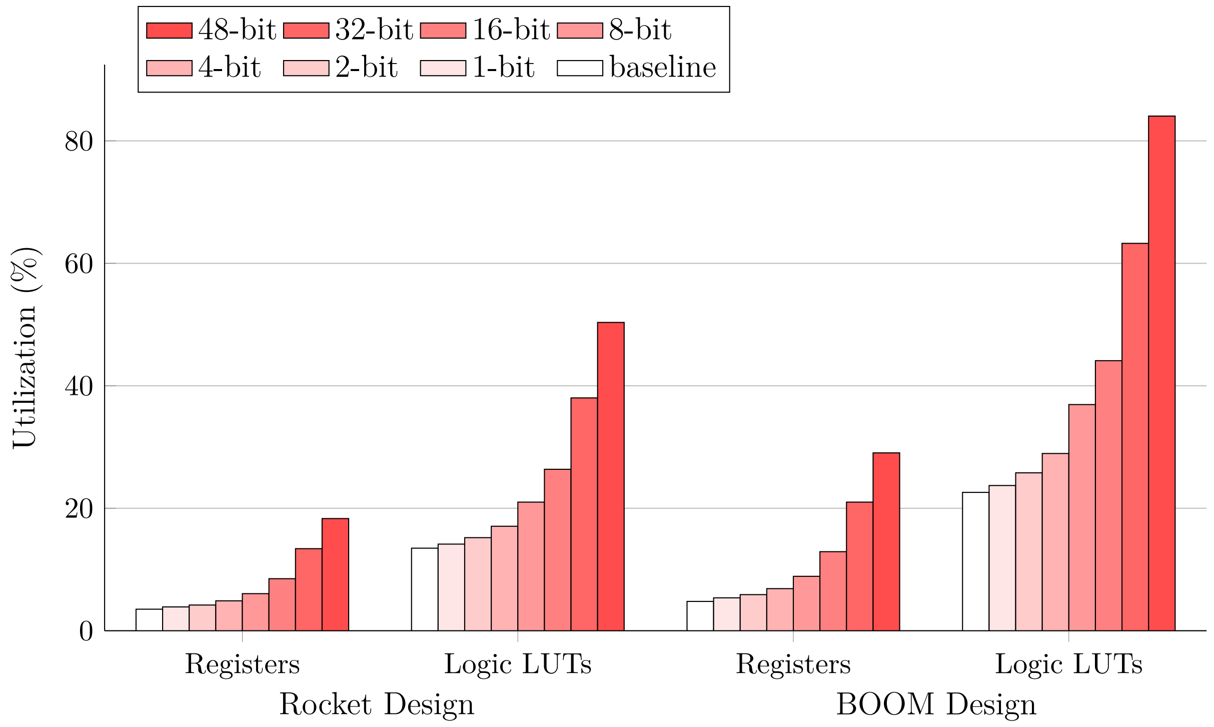


Figure 3.9: FireSim simulator FPGA resource utilization versus counter width on two different processor designs.

(65 MHz). Scanning out the 8060 cover counts at the end of the simulation took 12 ms. For the BOOM design the simulation executed 1.7 B cycles in 42.6 s (40 MHz). Scanning out the 12059 cover counts at the end of the simulation took 17 ms. In the future, we might be able to trade off simulation time and FPGA resource usage by using smaller counters that are sampled more frequently.

## Coverage Merging and Removal

Adding full line coverage to a large SoC design can have a significant area impact if we want to obtain high-resolution coverage counts. Also, note that mapping a FireSim simulation to the FPGA can take multiple hours. We can take advantage of the fact that we use the same coverage instrumentation for both FPGA and software-based simulation to filter out coverage points already caught in software simulation.

After merging the coverage results from running a RISC-V test suite with Verilator, we were able to reduce the number of coverage counters by 42 % by excluding the ones that were covered at least 10 times by the tests. As shown in Figure 3.9, resource consumption is dominated by coverage hardware for wide counters, with LUT utilization increasing by 2.8× in the 32 bit case. Once redundant points are removed, this falls to 2.0×, a tremendous saving that could be further improved with a more comprehensive suite of initial tests.

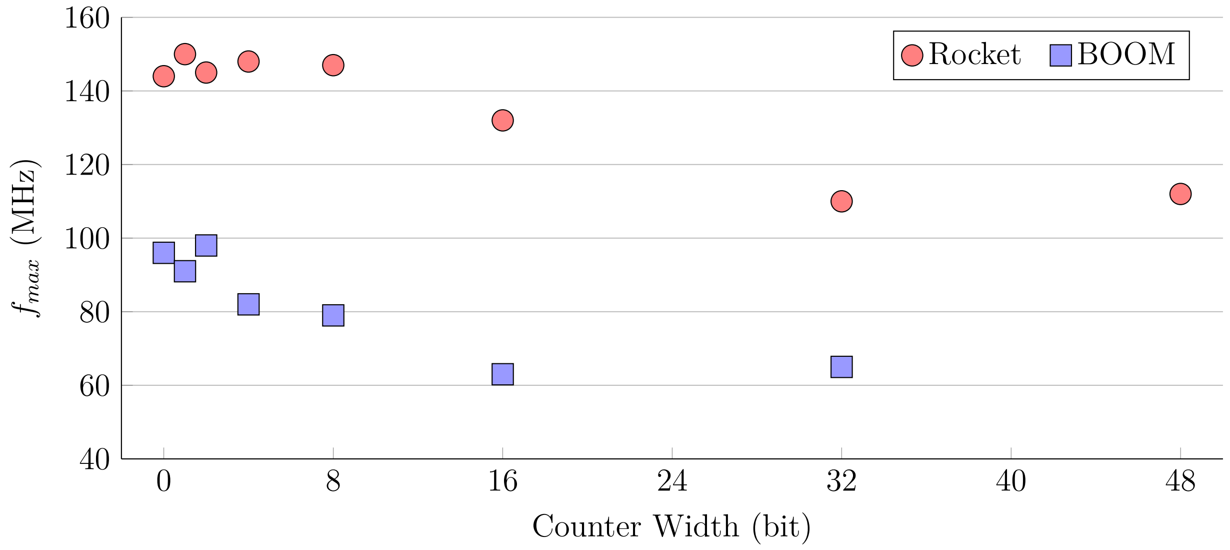


Figure 3.10: FireSim simulator $f_{max}$ versus counter width. A bit width of zero represents the baseline with no coverage support. Note, the 48 bit BOOM configuration did not place due to resource limitations.

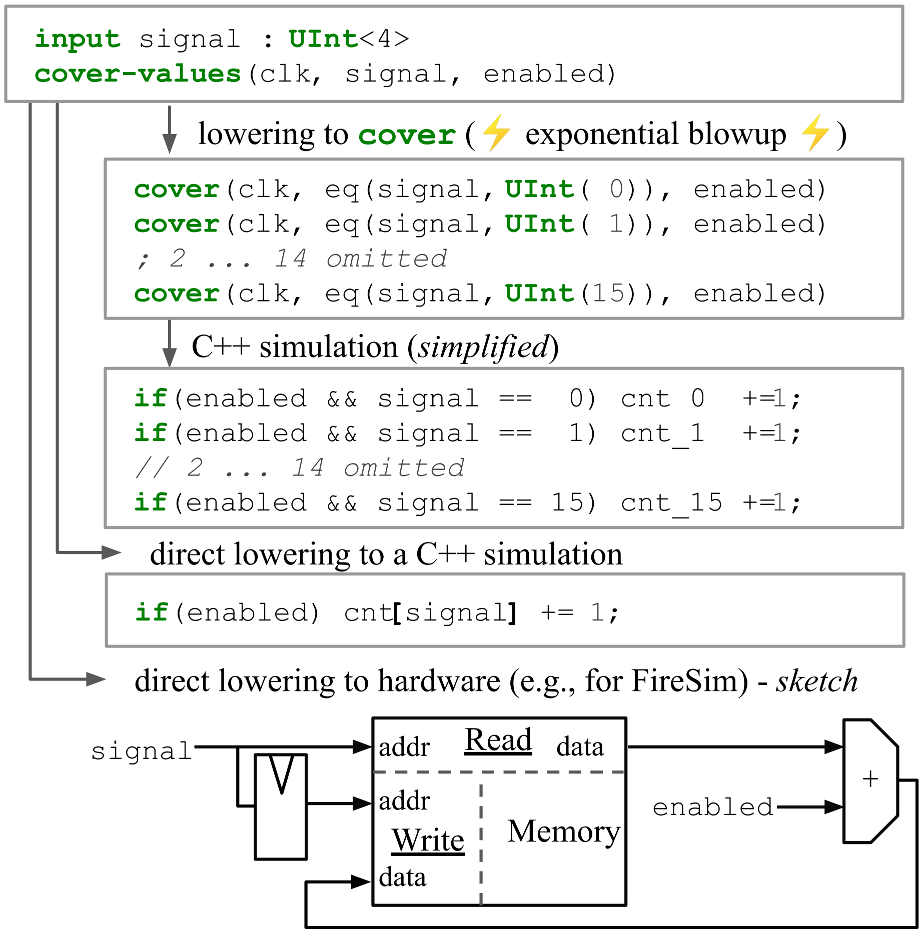
### Formal Trace Generation

As explained in Section 3.1 we can use our automated coverage instrumentation together with a formal tool to automatically generate traces that exercise our `cover` statements. We instrumented the open-source RISC-V Mini processor core and used bounded model checking to find `cover` points that cannot be reached in 40 cycles. RISC-V Mini was not a design that we were previously familiar with. Using formal trace generation with our line coverage instrumentation, we discovered that the RTL for the instruction and data caches are the same, but the instruction cache is read-only, and thus, the code blocks for write accesses are never exercised. When we used finite state machine coverage, we discovered a bug in our FSM analysis pass that resulted in an overestimate of transitions in the FSM. Formal verification revealed that these transitions could never be covered.

Thus, by moving the coverage instrumentation out of the simulator and into the FIRRTL compiler we are able to expand it to new use cases, such as automated coverage generation with a simple formal tool. This allows designers to explore their design easily and can also be very convenient for finding bugs in coverage instrumentation passes.

## 3.4 Limitations

We show that the most common types of coverage can be represented using synthesizable constructs and a `cover` statement. However, while working on this project, we uncovered one limitation that we would like to share: In the special case where we have a large number



```
input signal : UInt<4>
cover-values(clk, signal, enabled)
```

lowering to cover (⚡ exponential blowup ⚡)

```
cover(clk, eq(signal, UInt( 0)), enabled)
cover(clk, eq(signal, UInt( 1)), enabled)
; 2 ... 14 omitted
cover(clk, eq(signal, UInt(15)), enabled)
```

C++ simulation (*simplified*)

```
if(enabled && signal ==  0) cnt 0  +=1;
if(enabled && signal ==  1) cnt_1  +=1;
// 2 ... 14 omitted
if(enabled && signal == 15) cnt_15 +=1;
```

direct lowering to a C++ simulation

```
if(enabled) cnt[signal] += 1;
```

direct lowering to hardware (e.g., for FireSim) - *sketch*
signal
addr
Read
data
addr
Write
data
Memory
enabled
+

Figure 3.11: Covering all signal values with the `cover` statement leads to an exponential blowup. A `cover-values` statement could be lowered directly to significantly more efficient software and hardware implementations.

of events that we know are mutually exclusive, i.e., only one of them can occur in any given cycle, the use of multiple `cover` statements is sub-optimal since we cannot exploit the fact that only one of the counters will need to be incremented each cycle. A good example is that we might like to count how often a signal's value falls into certain cover bins. Implementing these cases efficiently requires a new `cover-values` primitive which counts how often a signal takes on each possible value. `cover-values` can be implemented in software by indexing into an array of counters or using a block RAM on the FPGA. This optimization becomes important when we want to cover a wide range of values, like in some fuzz testing applications [60]. Figure 3.11 demonstrates the exponential blowup when trying to use our `cover` statement and sketches efficient software and hardware implementations of `cover-values` inspired by prior work [60].

## 3.5 Discussion

Modern hardware verification flows rely on various simulators, emulators, and formal verification tools. This chapter demonstrates how a compiler-centered approach that lowers automated coverage metrics to a single primitive allows for uniform coverage support across backends. Support for a new simulator can be added in as little as a single day of work, and - by design - every coverage metric will be available from the start. We demonstrate coverage support for the Verilator and ESSENT simulators, which are significantly faster than many commercial tools [131] as well as the FPGA accelerated FireSim simulator, which simulates a four-core SoC at 65 MHz effective target frequency. In contrast, commercial emulators are generally known to be significantly slower.

Besides broad support for all coverage metrics, our technique enables features that would be difficult to support with a monolithic design where every coverage metric is hard-coded into the simulator. We can use a formal tool to generate coverage traces for all automatic metrics, including custom user-defined metrics like our ready/valid coverage. We can use coverage from a software simulation of a design to remove easily reachable cover points before instrumenting an FPGA-accelerated simulation with coverage counters.

# Chapter 4

# Coverage-Directed Fuzz Testing of RTL on FPGAs

In Chapter 3, we presented the `cover` statement abstraction, which allows us to quickly implement various coverage metrics for a wide variety of different simulators. The coverage numbers that are thus collected serve as valuable feedback to the RTL test designer. In this Chapter, we show how – instead of using coverage for humans to interpret – we can use a computer program to directly generate new test inputs based on the coverage feedback. Our solution is based on feedback-directed mutational fuzz testing, which was popularized for software testing by the AFL tool [156]. We were the first to adapt this technique to RTL testing and to demonstrate the feasibility of fuzzing a design while it is being simulated on an FPGA when we published at ICCAD in 2018 [81].

When the coverage feedback is used to drive the input generation, this problem is known as *Coverage Directed Test Generation* (CDG). Various solutions have been proposed over the last two decades. However, we argue that they are either designed for a very narrow class of DUTs or require a good amount of expert time, such as for constructing a DUT-specific Bayesian network [49]. This might explain why generator-based approaches, which require the test engineer to manually specify biases from coverage reports, are still the most widely used technique today.

Over the last couple of years, coverage-guided mutational fuzz testing has emerged as one of the most effective testing techniques for finding correctness bugs and security vulnerabilities in real-world software systems [156]. This technique relies on the fact that many interesting programs can be run quickly with arbitrary bytes as input. The program under test is augmented with lightweight instrumentation that provides feedback on the coverage achieved by a particular input. Starting from one or several seed inputs, the fuzz engine tries to achieve new coverage by mutating previously discovered inputs. Once a new interesting input is discovered after running the program under test on it, it is added to the input pool to serve as a new starting point in the input space exploration. Compared to more formal techniques such as symbolic execution, fuzz testing has been able to scale up to much bigger real-world programs with a smaller setup and engineering effort.

We believe that coverage-guided mutational fuzz testing is a new and interesting design point for solving the CDG problem. The approach of treating the test input as a series of bits or bytes allows this technique to be applied to a wide range of different circuits. Using FPGA-accelerated RTL simulation and synthesizable coverage feedback, we can achieve a high test execution speed similar to how some clever engineering techniques enabled fast fuzz testing speeds for software. In this regard, fuzz testing is — to the best of our knowledge — the first CDG technique to be designed specifically with FPGA emulation in mind [105].

In this chapter, we lay the groundwork for applying coverage-guided mutational fuzz testing to the CDG problem: We define the test stimuli in such a way that mutation algorithms from software testing can be directly applied. We solve the problem of deterministic test execution in FPGA-accelerated simulation by introducing transformations for `MetaReset` transformation and Sparse Memories. We define the notion of *Mux Toggle Coverage* that can be acquired during FPGA-accelerated simulation and used as feedback to the fuzz testing process. We empirically evaluate the performance of the proposed solutions on a variety of real-world RTL circuits ranging from communication peripheral IPs to CPU cores. Finally, we make our high-performance implementation of the proposed testing approach available to the research community as open-source software that can easily be used on a public cloud infrastructure for FPGA-accelerated fuzz testing experiments.

## 4.1 Coverage-Directed Mutational Fuzz Testing

**Algorithm 1** Coverage-guided mutational fuzzing

**Given**: program $p$, set of initial inputs $I$
**Returns**: a set of generated test inputs

```
1:  $\mathcal{S} \leftarrow I$
2:  $totalCoverage \leftarrow \emptyset$
3:  repeat
4:      for $input$ in $\mathcal{S}$ do
5:          for $1 \leq i \leq$ NUMCANDIDATES($input$) do
6:              $candidate \leftarrow$ MUTATE($input, \mathcal{S}$)
7:              $coverage \leftarrow$ RUN($p, candidate$)
8:              if $coverage \not\subseteq totalCoverage$ then
9:                  $\mathcal{S} \leftarrow \mathcal{S} \cup \{candidate\}$
10:                 $totalCoverage \leftarrow totalCoverage \cup coverage$
11: until given time budget expires
12: return $\mathcal{S}$
```

In this section, we introduce the basic *coverage-directed mutational fuzz testing* components and algorithm as used by the popular software fuzzer AFL [156] and various work that builds on top of it. A coverage-directed mutational fuzz testing tool (fuzzer) consists of three

Table 4.1: Deterministic mutation techniques.

| Name | Description |
|---|---|
| bitflip 1/1 | flip single bit |
| bitflip 2/1 | flip two adjacent bits |
| bitflip 4/1 | flip four adjacent bits |
| bitflip 8/8 | flip single byte |
| bitflip 16/8 | flip two adjacent bytes |
| bitflip 32/8 | flip four adjacent bytes |
| arith 8/8 | treat single byte as an 8-bit integer, add/sub values from 0 to 35 |
| arith 16/8 | treat two adjacent bytes as 16-bit big/little endian integer, add/sub values from 0 to 35 |
| arith 32/8 | treat four adjacent bytes as 32-bit big/little endian integer, add/sub values from 0 to 35 |

components: (1) A fuzz server that snapshots the program under test and quickly resets it before every test. (2) A static or dynamic instrumentation pass that augments the program under test to provide feedback about its behavior during execution. (3) A fuzz engine which implements the algorithms to select parent inputs, mutate them, and analyze the feedback from the instrumentation.

The program under test (PUT) is modeled as a pure function that takes an arbitrary length byte array as input. The behavior of the PUT should only depend on the selected input, and thus, a given input precisely describes a test execution. In order to guarantee that all test results are reproducible, it is imperative that all program state is reset in between tests. The fuzz server uses memory snapshot techniques to perform the reset before rerunning the PUT with a new input provided by the fuzz engine.

The goal of the instrumentation pass is to augment the program so that it provides coverage feedback for every execution. The feedback is used by the fuzz engine to guide its search of the input space and to create a test corpus with high coverage. In order to be effective, the chosen coverage metric needs to be lightweight enough to not slow down test execution significantly, detailed enough to be able to guide the fuzz engine, but also abstract enough to not overburden the fuzzer with every little detail of the PUT behavior. The coverage feedback used by AFL that has shown to be successful in practice for software testing is an approximate version of branch coverage: During instrumentation, every basic block in the PUT is assigned a random ID. Once the program takes a transition in the control flow graph, the source and destination ID are hashed together and used to index into a 65536 entry table of 8-bit counters. The selected counter entry is then incremented. In a post-processing step, the counter values, which range from 0 to 255, are placed into 8 exponentially increasing buckets. The intuition behind this is that traversing an edge twice instead of once is new and interesting, whereas going from 6 to 7 transitions is normally not relevant. This technique has good accuracy for small to medium-sized programs, is relatively

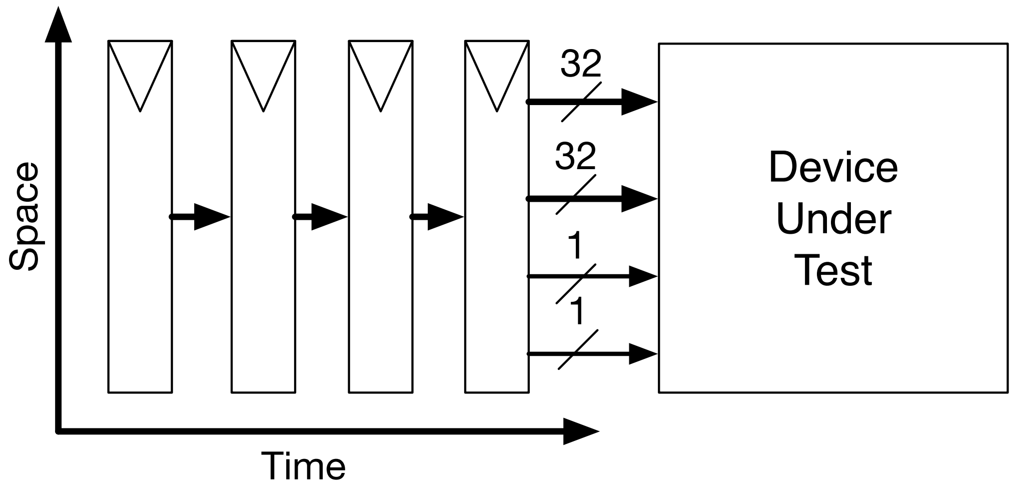


Figure 4.1: Input definition.

easy to implement, and has an acceptable performance overhead.

At the core of a fuzz testing system, the fuzz engine is responsible for selecting new test inputs to be evaluated and analyzing the resulting coverage feedback from the PUT. The algorithm (shown in Algorithm 1) starts with an initial set of *seed* inputs, e.g., a set of small PNG images when testing a PNG parser. Sometimes, a single empty input is used as a starting point. All seed inputs are placed in the test set data structure $S$. In the main loop, the fuzz engine selects one input from $S$ and applies a set of mutations to it. Each result of a mutation is executed by the fuzz server and the resulting coverage is analyzed. If a new coverage point is reached, the input that caused it is added to $S$. After a user-controlled timeout, the fuzzing process terminates, and the inputs in $S$ can be used as a test corpus.

The mutation algorithm uses two kinds of mutators: deterministic and non-deterministic. A mutator is a function that takes a test as input and modifies it in order to generate several new *child* tests. An example of a deterministic mutator in AFL is the `bitflip 1/1` mutation, which generates one child per bit in the parent input with the corresponding bit inverted. A list of all deterministic mutators that are relevant to this chapter can be found in Figure 4.1. The non-deterministic mutations are performed in the so-called `havoc` stage of AFL. In each application of the `havoc` mutation, between 2 and 128 random mutations are performed on the parent input. A list of all possible submutations is presented in Figure 4.2. While deterministic mutations mutate every position in the input, non-determinist mutators randomly choose the position to mutate.

The main advantages of coverage-directed mutational fuzzing compared to more formal techniques such as symbolic execution are the smaller engineering effort, better scalability to large real-world programs such as web browsers, and the portability of the approach due to its relative simplicity. The generated test corpus contains inputs that tend to be small. If a

Table 4.2: Non-deterministic `havoc` mutations.

| Name | Description |
|---|---|
| bitflip | flip a random bit |
| interest 8 | overwrite a random 8-bit integer with interesting value |
| interest 16 | overwrite a random 16-bit integer with interesting value |
| interest 32 | overwrite a random 32-bit integer with interesting value |
| arith 8/8 | treat random single byte as an 8-bit integer, add/sub one value from 0 to 35 |
| arith 16/8 | treat two random adjacent bytes as 16-bit big/little endian integer, add/sub one value from 0 to 35 |
| arith 32/8 | treat four random adjacent bytes as 32-bit big/little endian integer, add/sub one value from 0 to 35 |
| random 8 | overwrite random byte with random value |
| delete | delete a random sequence of bytes |
| clone | clone a random sequence of bytes |
| overwrite | overwrite a random sequence of bytes |

bug is uncovered while fuzzing (e.g., if a program crash is observed), the test input serves as a witness of the bug and can be used to debug the problem. The simple input definition of an array of bytes works on a wide range of programs and the coverage feedback is carefully engineered for good test performance.

## 4.2 Fuzz Testing of RTL Circuits

While coverage-directed mutational fuzz testing is an input generation technique that uses coverage feedback, it cannot be directly applied to the CDG problem for hardware designs: A digital circuit is not a binary file format parser that can read an arbitrary number of bytes. Instead, it has a number of input wires that can take different values in each cycle. In addition to that, the memory snapshotting techniques used to reset software to a known state before each test [156] cannot be directly applied to FPGA-accelerated RTL simulation. Instead, we need a way to quickly reset the RTL state without changing the behavior of the DUT. Furthermore, while its notion exists in HDLs for RTL simulation, branch coverage does not directly apply to RTL designs. Branches in the HDL source code are mapped to multiplexers in the circuit which output one of two input values during each cycle, which is different from sequential software where only one branch is active at a given point in time. In this section we therefore discuss how a test input for RTL circuits can be defined, the work necessary to make DUTs resettable on the FPGA, and how the notion of branch coverage can be translated to testing RTL circuits.

## Input Definition

RTL circuits are commonly represented as module hierarchy featuring one top-level module that will be connected to the test harness or external pins. In our methodology, the top-level input pins are connected to the testing tool. We concatenate all input pins and map the resulting bit vector to a series of bytes representing the input values in one particular test cycle. To allow the fuzz engine to apply a different value during each test cycle, we concatenate single-cycle test inputs to form a multi-cycle input. We thus concatenate inputs in space and time as illustrated in Figure 4.1. The number of cycles a particular test runs for is thus determined by the number of *test input* bits divided by the number of input bits to the top-level module of the DUT. Since the DUT is reset to a known state before each test execution, the test inputs fully describe the test execution. Reproducing coverage or assertion violations thus only requires knowledge of the DUT and the test input.

## Deterministic Test Execution

Fuzz testing requires the program or device under test to be started from a known state to make tests deterministic and repeatable. This ensures that only the test inputs affect the behavior that will be observed by the fuzz engine and thus, a test can be fully reproduced as long as the inputs are known. Resetting a program to a known state can require a non-trivial amount of time. The simplest approach, to restart the program under test for every test invocation, includes the cost of loading the program into memory and process creation, which limits the number of test executions per second. The popular fuzzer AFL takes advantage of the `fork` system call and copy-on-write optimization by the operating system to reduce the overhead.

Quickly resetting an RTL circuit mapped onto an FPGA poses its own set of challenges which need to be addressed in order to apply fuzz testing to this domain. In the following sections, we discuss two major problems and how they are solved in our work: (1) for efficiency reasons, many registers are not reinitialized during device reset; (2) memories do not feature reset circuitry and can only be initialized one word at a time

## Register Meta Reset

Reset circuitry for registers takes up space on the wafer and is thus omitted from designs whenever possible. The initial value of these registers is thus undefined when the DUT comes out of reset. Classic circuit simulators deal with this fact by introducing an $X$ value, which marks an uninitialized wire (in 4-state simulation) or by randomizing the initial values of the register (in a 2-state simulation). Since this work is targeted at FPGA-accelerated simulation, we would like to make use of a 2-state solution in order to prevent the blowup in size that would result from emulating a 4-state simulation. Randomizing the register values on the FPGA, however, is also a non-trivial endeavor and could lead to sporadic tests. Instead, there are two promising solutions: (1) we can treat the initial register state

```
reg [31:0] r;
always @(posedge clk) begin
  if (reset) begin
    r <= 32'h1993;
  end else begin
    r <= r_next;
  end
end
```

(a) Register With Reset

```
reg [31:0] r;
always @(posedge clk) begin
  if (metaReset) begin
    r <= 32'h0;
  end else begin
    if (reset) begin
      r <= 32'h1993;
    end else begin
      r <= r_next;
    end
  end
end
```

(b) Register With MetaReset

Figure 4.2: Meta reset transformation.

as part of the input and load the values through a scan chain before each test (2) we can reset all registers to a predefined value before each test.

In this work, we implement solution 2 in a transformation pass that works on the intermediate representation (IR) of the circuit and adds a `MetaReset` wire, which resets all registers in the circuit to zero. As an example, Figure 4.2 illustrates the addition of a `MetaReset` to a register description in Verilog. Our test harness thus applies the following sequence before each individual test: First, the `MetaReset` is activated for one cycle in order to initialize each register to zero. Next, the `MetaReset` is released and the actual `Reset` of the DUT is asserted in order to take the device through the reset procedure envisioned by its designer. During this phase, it is essential to provide deterministic inputs to the DUT because a register might be hardwired directly to a top-level input in the original design. We again chose all zeros as a deterministic input. This approach is sound since starting the design with all registers set to zero is allowed by the RTL (an $X$ can be any value), but incomplete since other possible values are not explored.

## Sparse Memories

Memories are rarely meant to be reset when the circuit is turned on. Memories are often mapped to SRAM cells, which generally contain arbitrary values when powered on. The concept of a memory can be mapped to a much more area-efficient implementation compared to a register because only a small number of memory words (bounded by the number of write ports) can be updated in each cycle. This restriction sets the lower bound for the number of cycles needed to fully reset a memory to be the memory size in words divided by the number of write ports. This number can be very high in practice, especially when considering the data or program memory of a processor design.

The task of resetting memories is not as difficult as the upper bound mentioned above might suggest. To see why it is important to consider the details of the fuzz testing scenario proposed in this chapter: The core idea is to mutate the seed inputs as often as possible and to evaluate every generated input on the instrumented DUT. In order for this to be feasible, the test size and thus, the number of cycles it takes to execute a given test is relatively small. Thus, the number of *writes* that may occur during a single test for a given memory is bound by the number of write ports times the number of test cycles. If we can keep track of the memory locations that have been written in a test execution, we are able to undo all the changes made and thus reset the DUT on the FPGA. This solution would require a memory for every write port to remember the addresses that have been written to in addition to the actual memory that stores the values. In the worst case, resetting this kind of memory would take as many cycles as the previous test took to execute.

Going back to the observation that we will only observe a small number of writes, we can refine the design of our resettable memory: Since the number of writes is small, most memory locations will contain the reset value which we define to be zero, just as for registers. The read port of such a *sparse memory* needs to work in the following manner: If the address that is requested has been written to, return the last written value. If the address that is requested has never been written to, it is uninitialized, and thus, we need to return zero. We can keep track of the addresses that have been written to, by using a content addressable memory (CAM). The CAM can also be used to map the requested address to a much smaller memory that only needs to be able to hold as many values as may be written during a maximum-length test. Thus, this kind of *sparse memory* needs oftentimes much less SRAM than the original version. The CAM can easily be reset in a single cycle by connecting the valid bit of each entry to the `Reset` signal of the DUT. Implementing a custom transformation pass on the RTL of the DUT, we can automatically replace memories by a sufficiently large *sparse memory*.

Using the *MetaReset* and the *SparseMem* transformations, we can ensure that the DUT can be reset to a fully deterministic state in only two clock cycles. This allows for rapid and repeatable test execution, thus enabling fuzz testing of RTL circuits on FPGAs.

## Coverage Definition

We require precise definitions of our coverage metrics for two reasons: (1) To define an end-to-end metric that can be used to measure how well our implementation of mutational fuzz testing for RTL compared to a baseline technique; (2) To define an intermediate coverage metric that serves as feedback to the fuzz engine in order to guide the search of the input space. While prior industrial work [138, 145, 19] uses functional coverage models manually specified by verification engineers, these test suites are expensive to create and, thus, generally unavailable to the broader research community. Instead, we focus on automatic coverage that can be derived directly from our suite of open-source benchmark circuits. This kind of coverage has been used in related academic [133] and industrial [65] work.

Most automatic coverage definitions focus on the description of the circuit expressed in a common HDL like Verilog or VHDL [117]. However, our system works with any RTL circuit regardless of the hardware description or generation language it is written in. We thus define our coverage metric in relation to the synthesizable structure of the circuit as represented in an HDL-agnostic IR. This also ensures that we can synthesize and thus collect coverage information during FPGA-accelerated simulation.

We look at *mux control coverage* which treats each 2:1 multiplexer select signal as an independent cover point. Multiplexers with more than two inputs can be trivially converted into a series of 2:1 multiplexers. We chose this metric as it can be automatically applied to any RTL circuit as long as the multiplexers are explicitly modeled. It is also well-suitable for FPGA-accelerated simulation since we do not need any additional circuitry to evaluate the cover points.

For a mux control condition to be fully covered, we require that it evaluates to true as well as to false during a single test. On the FPGA, this requires only minimal additional hardware — two 1 bit registers and two 1 bit multiplexers — to remember the observed values. We combine the coverage observed in multiple tests by calculating the union of mux control conditions covered. Note that by this definition it is not enough for a condition to always be true in one test and always be false in another test to be counted as covered. Instead, both values need to be observed in a single test.

The *coverage* used in Algorithm 1 is thus defined to be the set of multiplexers in the DUT which had their control signal toggle during test execution. If a test input manages to toggle a mux control signal that had never been toggled before, it is considered interesting and is added to the test data structure $S$.

## Mutation Algorithms

Our input definition allows us to directly implement the mutation heuristics from the successful AFL fuzz testing tool [156] presented in Section 4.1. Similar to AFL, every new entry in our test set is first mutated with the deterministic mutation techniques listed in Table 4.1. Once we run out of deterministic mutations to apply, we switch to our implementation of the AFL `havoc` stage which makes use of the mutations presented in Table 4.2. For any mutation that changes the size of the input array, we pad with zeros when necessary in order to maintain an input size that is a multiple of the bytes needed in a single cycle.

## Constrained Interfaces

In hardware testing, many interfaces make assumptions that have to be respected by the stimuli generator. An example for this is a memory bus which can rely on the fact that any participant will respect the protocol specification. To this end, a test input generator in traditional directed random testing needs to be implemented in such a way as to not violate the guarding assumptions. A similar solution could be applied to fuzz testing by implementing an RTL adapter that takes the unconstrained inputs from the fuzzer and — by construction — generates valid bus transactions from them.

However, the feedback-directed manner of the fuzzing approach allows for a more convenient solution: We observe that modern HDLs allow designers to specify interface constraints through `assume` statements in the DUT source code. In our benchmarks which make extensive use of the TileLink bus, this mechanism is used to implement a synthesizable bus monitor which detects invalid transactions. Taking the conjunction of all assumptions in the monitor over all cycles in a test, we can derive a binary signal which indicates whether the given test inputs exercise the DUT in a valid manner. This *valid* signal is included by the test harness on the FPGA with the regular mux condition coverage as feedback to the fuzz engine. The simplest way of using this signal is to reject all invalid inputs before updating the coverage map. This is comparable to rejection sampling in random directed testing.

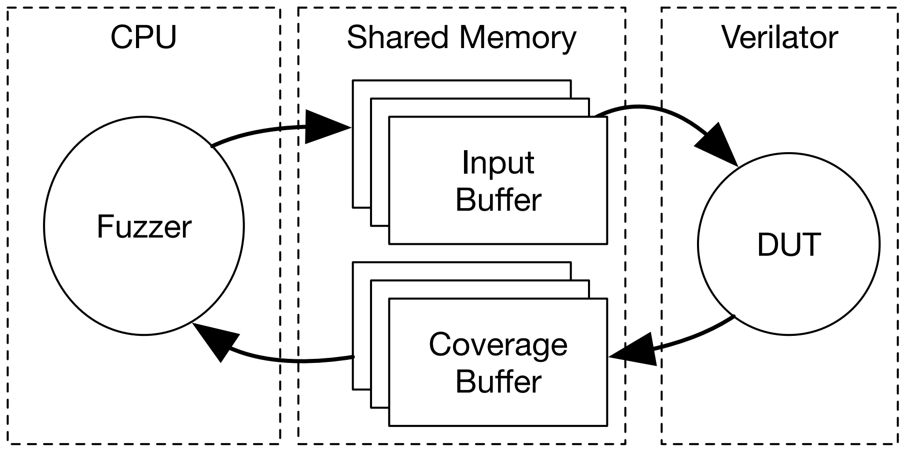
The authors of the open source Java fuzz testing tool JQF [1] have extended the core fuzz testing method described in Algorithm 1 to take advantage of the feedback regarding the validity of a generated test input. They keep two separate coverage maps: one for the total coverage and one for the coverage achieved by valid inputs only. A new input is added to the test set $S$ when it achieves new total coverage or if it is valid and achieves new valid coverage. This extension allows the fuzz engine to discover valid inputs from invalid ones. We implemented the JQF technique using two coverage maps in our testing tool.

## 4.3 Implementation

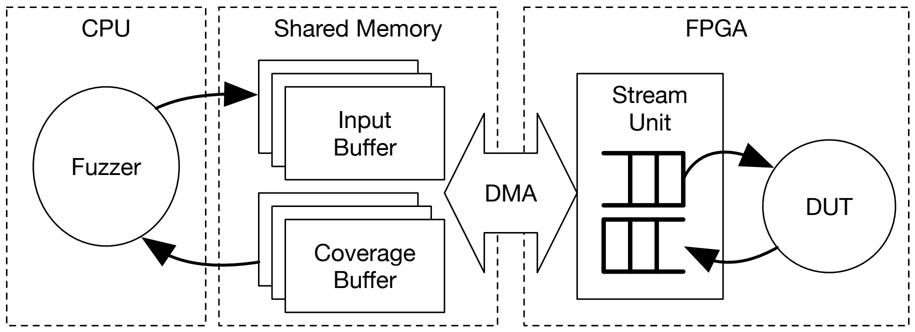
We implemented the proposed testing methodology in an open-source tool called RFUZZ[2]. It consists of an instrumentation and harness generation component that works on arbitrary RTL circuits described in the FIRRTL IR [67]. The first part of our tool is an instrumentation and harness generation component, which works on arbitrary RTL circuits described in the FIRRTL IR [67]. It automatically generates a test harness for software or FPGA-accelerated simulation. The second part of our tool is the actual input generator, which connects to the test harness running in software or on the FPGA to provide DUT inputs and analyze the resulting coverage.

Our tool is language-agnostic since it can work on arbitrary RTL designs expressed in the FIRRTL IR [67]. Once a target design is translated into FIRRTL IR from its source HDL, we can apply compiler passes for the target RTL regardless of its source HDL. RFUZZ is also fully automated as the target RTL is instrumented through compiler passes, and the fuzzer uses the target information generated by the compiler. Only some parameters to the fuzzer, such as the mutation technique and seed inputs to use, need to be specified by the user.

[1]`https://github.com/rohanpadhye/jqf`
[2]`https://adept.eecs.berkeley.edu/papers/rfuzz`



(a) Fuzzer with software simulation.



(b) Fuzzer with FPGA-accelerated simulation.

Figure 4.3: Shared memory implementations for communication between the fuzzer and the test harness.

## Instrumentation

Custom transforms are implemented as compiler passes that plug into the FIRRTL compiler. We use the compiler's dead code elimination and constant folding to minimize the redundant expressions before instrumenting the coverage signals. This helps us keep the size of the automated coverage feedback as small as possible.

The *MuxCov* (Section 4.2) pass automatically identifies intermediate coverage wires by traversing the circuit description. In our implementation, we consider multiplexer control conditions, memory read and write enables, and memory masks for the coverage feedback to the fuzzer. Coverage wires automatically identified by the circuit traversal are then rewired through the module hierarchy to be available as outputs of the top-level module so that coverage wire values are observed by the test harness. This pass also generates a metadata file containing information about the coverage and the input pins in the RTL design for the

Table 4.3: Benchmarks.

| Name | Input Width | Mux Cover Points | Lines of FIRRTL |
|---|---|---|---|
| Sodor1Stage | 35 | 714 | 3617 |
| Sodor3Stage | 35 | 746 | 4021 |
| Sodor5Stage | 35 | 945 | 4088 |
| I2C | 165 | 301 | 2373 |
| SPI | 167 | 323 | 4046 |
| FFT | 259 | 195 | 1545 |
| Rocket Chip | 239 | 4517 | 43856 |

test harness generation.

The details of our *MetaReset* and *SparseMem* passes are explained in Section 4.2 and Section 4.2, respectively.

## Test Harness Generation

The test harness generator automatically generates a wrapper for any RTL design by consuming the target design information, including input and coverage pins generated, by the instrumentation passes. It instantiates the instrumented DUT and connects the coverage pins inserted by the instrumentation pass to toggle detection circuitry. It also automatically derives a buffer format definition for the required input and coverage size and emits Verilog and C++ code for the software and FPGA-accelerated simulation environments to interface with the buffers.

The test harness is further automatically transformed by FIRRTL compiler passes for efficient token-based simulation on the FPGA [76], dramatically reducing manual effort for FPGA-accelerated simulation. Our tool also automatically generates the buffer stream unit mapped on the FPGA and integrates it with the test harness for communication with the fuzzer. Finally, the test harness generator emits target-specific information about coverage counters, top-level inputs, and buffer formats, which the fuzzer consumes to test a particular circuit design.

## Fuzzer

Whereas the DUT and the coverage counters can be synthesized onto an FPGA, the input generation and coverage analysis are performed by a fast fuzzer on the CPU. Implementing this part in software allows for greater flexibility in investigating new mutation and feedback strategies. While an integrated solution on the FPGA could be even faster, we achieve good performance with a high bandwidth DMA channel.

Figure 4.3 shows how the fuzzer efficiently communicates with the test harness through shared memory buffers. The fuzzer is unaware of whether the test harness is run in software simulation or on the FPGA. The fuzzer allocates multiple buffers in the shared memory region, and test inputs and coverage feedback are batched to the buffers (Figure 4.3a). When the test harness is run in software simulation, the software simulator directly accesses these buffers. To cope with high round-trip latency between CPU and FPGA, when the test harness is run in the FPGA, these buffers are transferred through a high bandwidth DMA to the buffer stream unit that post-processes the data in the buffers (Figure 4.3b).

## 4.4 Evaluation

We evaluate the proposed testing methodology using our RFUZZ tool on a range of open-source RTL designs:

1. **TileLink Peripheral IP**: These consist of a SPI and I2C peripheral IP which are used in the commercial SiFive Freedom SoC platform [3]. They interface with the fuzzer through a TileLink port which includes a synthesizable bus monitor. The feedback from the monitor is used to ensure that only valid inputs are included in the reported coverage.

2. **FFT**: As an example of a DSP block, we use a FFT implementation produced by an open-source FFT generator [4].

3. **RISC-V Sodor Cores**: We selected three different educational RISC-V cores maintained by the LibreCores project [5]. In order to directly affect the executed instructions, we create a special fuzz testing top-level module which — instead of instantiating a scratchpad memory — directly wires the instruction memory interface to the top-level inputs. This allows our testing tool to act as the instruction memory and directly supply the core with instructions to execute.

4. **RISC-V Rocket Core**: In order to test the scalability of our approach, we use the RISC-V Rocket Chip [6] as our final benchmark. This 64-bit in order core is supported by industry and is able to boot the Linux operating system. Its size impacts the execution speed of software simulation and thus allows us to evaluate the benefits of an FPGA-accelerated simulation approach to the CDG problem.

A detailed list of the benchmarks is available in Table 4.3.

For our evaluation we use software simulation on the public AWS cloud infrastructure to quickly evaluate various configurations in parallel. Each fuzz testing run was performed on its own virtual core. Since several of the proposed mutation techniques make random

[3]https://github.com/sifive/sifive-blocks
[4]https://github.com/ucb-art/fft
[5]https://github.com/ucb-bar/riscv-sodor

Table 4.4: Speedup: FPGA vs software simulation.

| | Sodor3Stage | Rocket |
|---|---|---|
| Verilator | 345 kHz | 6.89 kHz |
| FPGA | 1.7 MHz | 1.46 MHz |
| Speedup | 4.9x | 212x |

Table 4.5: Machine specifications for speedup evaluations.

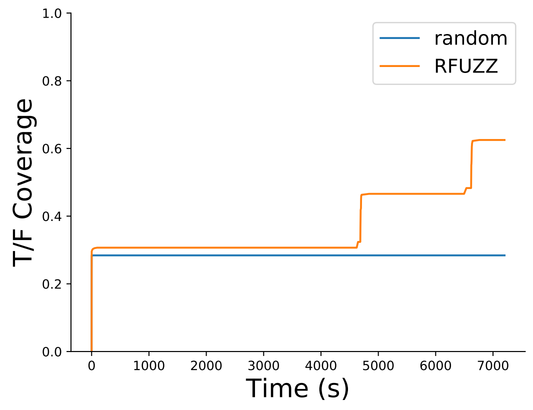
| | Local Machine (Verilator) | Amazon F1 (FPGA) |
|---|---|---|
| CPU | AMD Ryzen 7 1700X | 8 vCPUs |
| Memory | 32 GB | 122 GB |
| FPGA | - | Xilinx UltraScale+ VU9P |
| DMA bandwidth | - | 1.5 GB/s |

decisions when generating new test inputs, we rerun experiments four times with different seeds to the pseudo-random number generator and average the results.

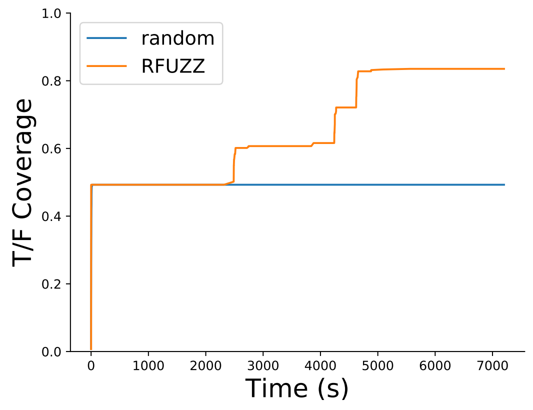
During each testing run, we save all generated inputs that make it into the test set $S$ to disk. In order to evaluate the achieved coverage independently from our testing tool, we use a series of Python scripts to calculate end-to-end coverage metrics. Since speed does not matter in this context, we are able to restart the software simulation for each entry can thus be confident that various tests are indeed independent and do not affect each other. The end-to-end analysis scripts also exclude any invalid inputs as indicated by an assumption failure during the test run. We can thus ensure that — independent from RFUZZ — our coverage numbers only include valid inputs as checked by the monitors and assume statements. All coverage in this section is measured as a fraction of the maximum mux control toggle coverage as indicated by the number of multiplexers in the design. It might be impossible for some mux control wires to be influenced from the inputs controlled by the fuzzer and thus there is no guarantee that 100 % coverage can be achieved.

## Comparison to Random Testing

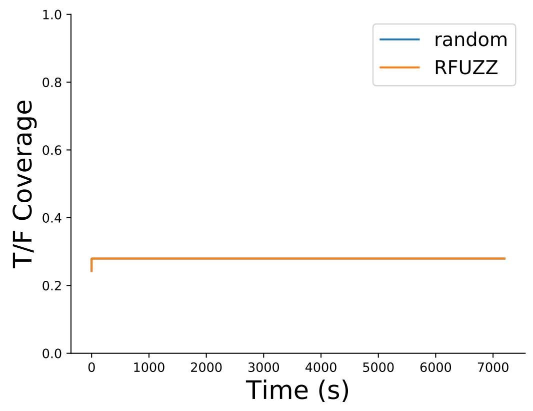
Similar to our proposed technique, random testing is applicable to any RTL circuit without DUT-specific setup costs. We implement random testing in our tool in order to measure whether coverage-directed mutational fuzz testing provides any advantages over the simple random baseline. In order to implement the baseline, we replace the normal mutation algorithms that modify a given test input to generate a new independent random input instead. Figure 4.4 shows the results for all of our benchmarks. As we can see, the random baseline quickly saturates, whereas the coverage-guided fuzz testing is able to make progress by mutating previously discovered inputs.



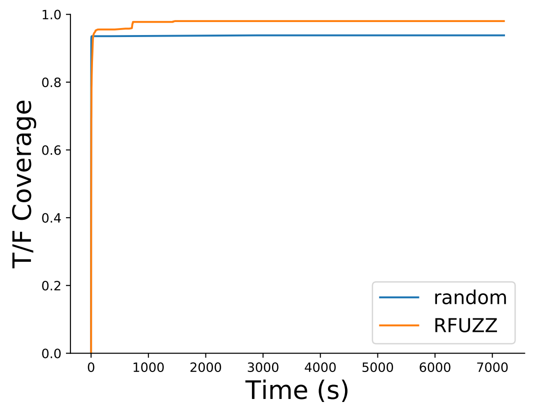
(a) I2C



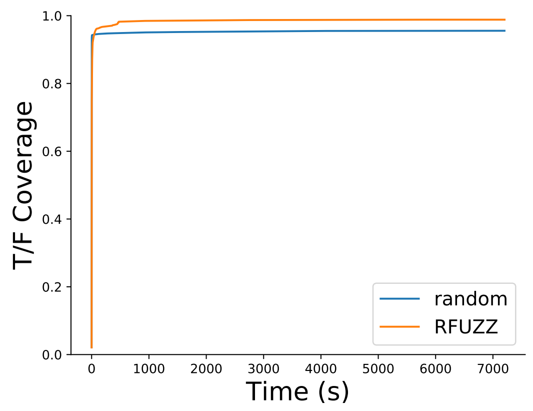
(b) SPI



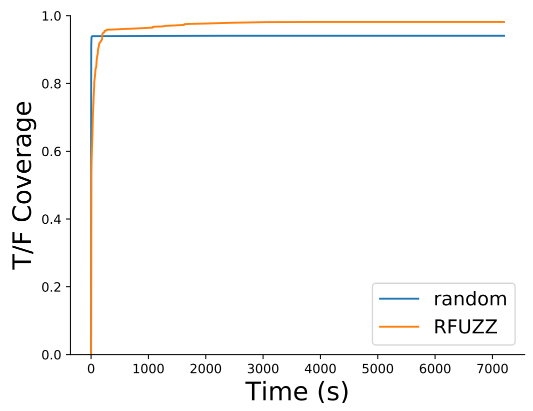
(c) FFT



(d) Sodor 1 Stage



(e) Sodor 3 Stage



(f) Sodor 5 Stage

Figure 4.4: Mux control toggle coverage over time: RFUZZ vs. random testing.

### Constrained Interfaces

As explained in Section 4.2 we can deal with constrained interfaces by observing the assumption failures that result from invalid test inputs. All tests in Figure 4.4 used the JQF technique to generate valid inputs. As we can see, this provides a significant improvement over the random baseline for the TileLink I2C and SPI peripherals.

### Software vs. FPGA-Accelerated Simulation

For small circuits, generating the test inputs and analyzing the resulting coverage takes the majority of time, but this changes as the design becomes bigger. The simulation time is the major bottleneck for large designs such as a real-world 64-bit processor. As mentioned throughout Section 4.2, we took specific care to design our testing methodology in such a way that the device under test can be simulated on the FPGA.

To show how FPGA-accelerated simulation enables us to scale to test complete large-scale systems, we measured the execution speed for a small educational processor (Sodor3Stage) and a productized in-order processor (Rocket). Table 4.5 shows the specifications for the machines we used in this evaluation. We compiled bitstreams for FPGA-accelerated simulations using Vivado 2017.1, and both designs closed timing at 75 MHz. The FPGA synthesis time was 2~5 hours.

Table 4.4 shows the speedup of FPGA-accelerated simulation over software simulation for two designs. As expected, we can achieve significant speedup for a complex design, but even a small design can benefit from FPGA-accelerated simulation. Notably, software simulation slows significantly with complex designs, while FPGA-accelerated simulation provides high simulation rates regardless of design complexities. With FPGA-accelerated simulation, the simulation speed is bottlenecked by the speed at which our fuzzing software analyses coverage and generates new inputs. Thus, shifting some of that functionality from the fuzzer to the FPGA could significantly improve the simulation rates in the future.

## 4.5 Advanced Coverage Metrics as Fuzzing Feedback

The work on coverage-directed mutational fuzz testing was done before we generalized the approach taken to implement mux-toggle coverage to the universal solution presented in Chapter 3. Thus, all our evaluations were limited to looking at mux-toggle coverage. However, with the `cover` statement-based approach, any metric that we implemented an instrumentation pass for can be used as fuzzing feedback. We thus created a simple fuzzing setup, connecting the AFL fuzzer [156] to a rfuzz-style harness [81] using the RTL Fuzz Lab infrastructure [48]. The coverage counts serve as direct feedback to AFL instead of going to a report generator. This way, we can mix and match various metrics easily.

We implemented the mux toggle coverage metric from rfuzz in our framework and compared it to using our line coverage as feedback when fuzzing an I2C peripheral. Figure 4.5

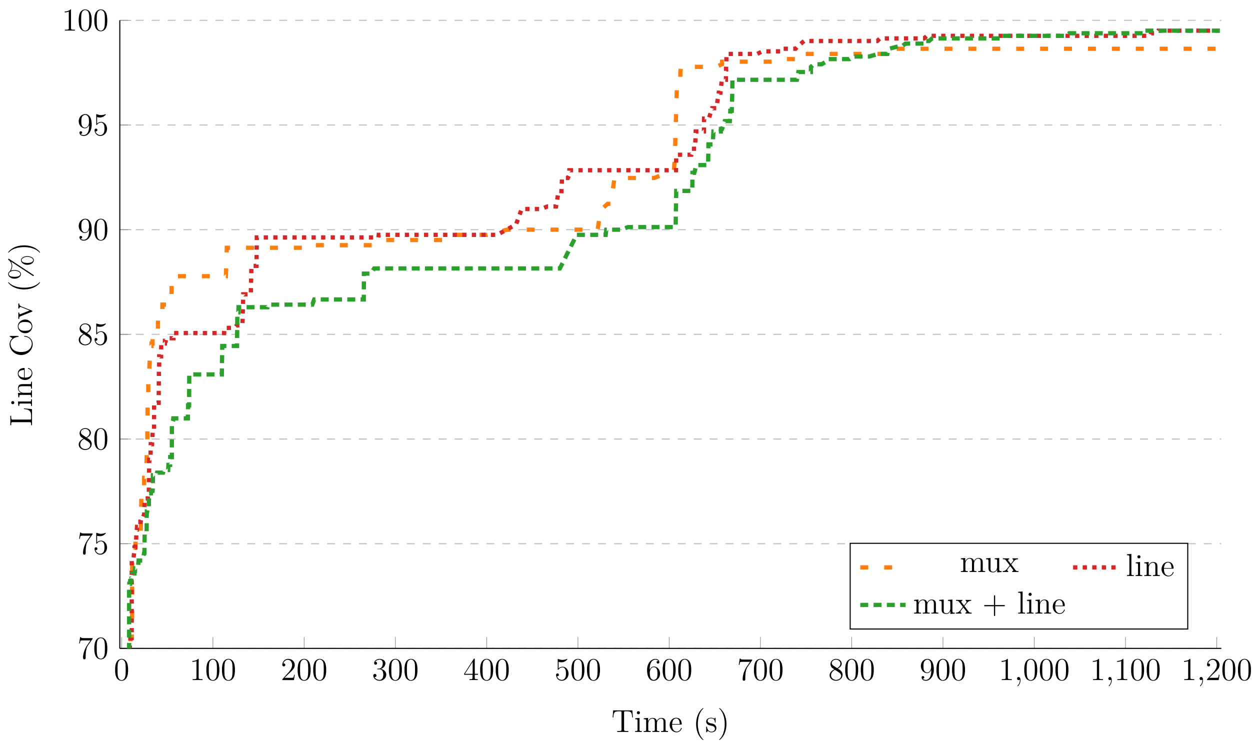


Figure 4.5: Cumulative line coverage of inputs to the I2C peripheral discovered through fuzzing with various feedback metrics. Averaged over five runs.

shows cumulative line coverage for different feedback metrics. We see that increasing mux-toggle coverage generally increases line coverage as well. Mux-toggle coverage can be implemented over low-level structural RTL, while line coverage generally works best on behavioral RTL. Thus, there may be situations in which it is best to use the mux-toggle coverage proxy. However, Figure 4.5 does show a slight advantage of using line coverage directly instead of a proxy metric when it comes to achieving maximum final coverage.

## 4.6 Discussion

In this chapter, we show how coverage-directed mutational fuzz testing can automatically test arbitrary RTL circuits on FPGAs. We provide a high-performance implementation of this technique based on the FIRRTL compiler and a new fuzzer implementation that decouples input generation from coverage analysis to effectively hide the latency inherent to communicating with a design under test on an FPGA. Our evaluation shows consistent improvements over random testing, especially for circuits that provide feedback on test input validity.

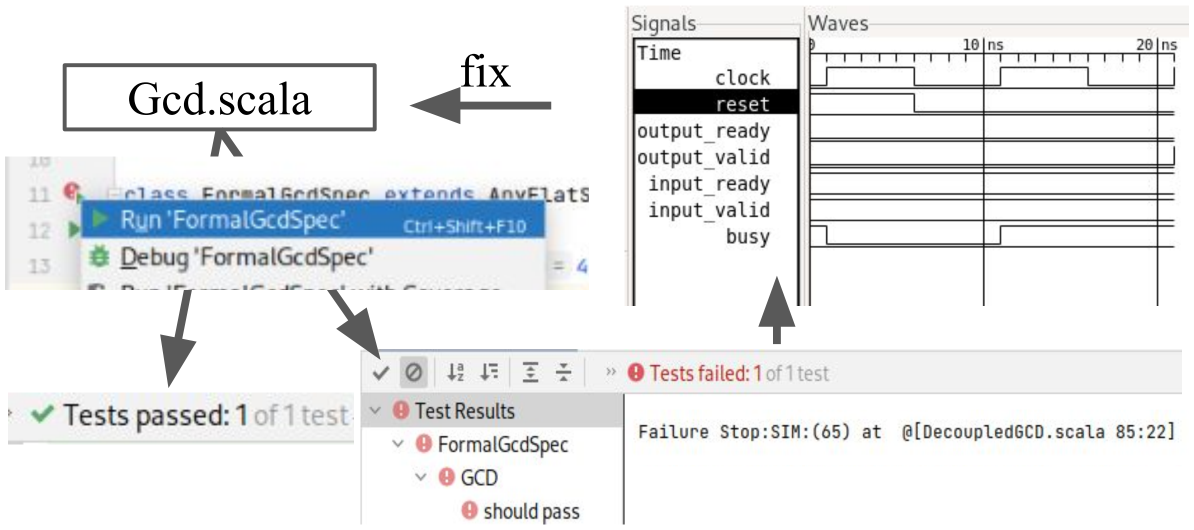
# Chapter 5

# Open-Source Formal Verification for Chisel

In Chapter 4, I demonstrated how feedback-directed mutational fuzzing can use coverage metrics to generate new and interesting inputs to a design under test. However, we might still miss corner case bugs even with diverse test inputs. An alternative approach, as introduced in Section 2.6, is formal verification. During my PhD, I extended the ChiselTest verification library with formal verification capabilities. In this work, I focused on providing a good user experience by integrating formal verification as tightly as possible with existing simulation-based testing infrastructure. The work described in this chapter has been published at the Workshop on Open-Source EDA Technology (WOSET) [78] and as part of a more extensive journal paper on ChiselTest-based verification [42]. All features have been part of the open-source ChiselTest library since the 0.5.0 release in 2021. I have used the formal verification support for my interactive guest lecture on formal verification for an agile hardware design course at UC Santa Cruz, which I was invited to give in 2022, 2023, and 2024.

There is a long tradition of open-source formal verification systems from the academic community [96, 99, 31, 114, 54, 147, 103]. However, because of the traditional academic incentive structure, these research systems are often complicated to use and lack support for advanced Verilog features, preventing them from being widely used by a community of open-source RTL designers. This has changed with the introduction of the yosys [153] tool, which has become the de facto standard for processing Verilog for synthesis or formal verification. Yosys allows academics to focus on developing model checkers for the simple btor2 [111] or aiger [14] formats without worrying about supporting the much more complicated Verilog standard. The open-source SymbiYosys [152] tool wraps yosys and various formal verification engines to allow users to verify their designs. All a user has to provide are the Verilog sources of their design, including assertions and assumptions, and a small configuration script. SymbiYosys translates any failing traces it discovers into Verilog test benches and VCD waveform dumps for the user to inspect. With the open-source GHDL plugin, yosys also supports formally verifying circuits written in the VHDL language.

In this chapter, we describe our approach to providing Chisel users with an easy way to



```
class FormalGcdSpec extends AnyFlatS
```

```
Failure Stop:SIM:(65) at  @[DecoupledGCD.scala 85:22]
```

Figure 5.1: When working in a standard Scala IDE like the open-source IntelliJ IDEA with the Scala plugin, the user can launch the formal check with the press of a button. The success or failure will be communicated like any other unit test. A VCD waveform dump is automatically generated to help debug failing checks.

formally verify their designs, similar to what SymbiYosys provides for RTL designs written in Verilog. We adopt many good ideas from yosys and build several new convenience features, taking advantage of the existing compiler infrastructure for Chisel. We added several novel Chisel-specific features that will make formal verification accessible to all Chisel users. Our system can automatically add reset assumptions, thanks to Chisel modules always having a known reset pin. We employ simulation replay to help users investigate failing inputs found with bounded model checking. We carefully designed a formal backend for the FIRRTL compiler to ensure that undefined values in Chisel are correctly modeled and that the model checker can explore all possible behaviors. We also use the extensibility of the Chisel library and FIRRTL compiler to implement a `past` expression for simple and safe temporal assertions.

## 5.1 Our Formal Verification Flow

Before we dive into some of the details of our implementation, we want to present the workflow that our tools enable from a user's perspective to illustrate how easy it can be to get started with formal verification of a Chisel circuit. The recommended way to start a Chisel project is to use the open-source Chisel template repository. The resulting Scala project automatically includes dependencies on the Chisel and ChiselTest libraries, which the Scala build tool downloads automatically for the user.

The template contains an example of using the ChiselTest library to test a greatest common denominator (GCD) circuit in simulation. The full source code of the circuit and its test can be found in the appendix in Listing 1 and Listing 2. The user can execute this test through their Scala IDE or from a shell with the `sbt test` command. To turn this test into a formal check, we just need to substitute the `test(new DecoupledGcd(16))` command with `verify(new DecoupledGcd(16),` as well as provide the type of verification job as `BoundedCheck(10)` and extend the testing class with the `Formal` trait:

```
class GCDSpec extends AnyFreeSpec with ChiselScalatestTester with Formal {
  "Gcd should verify" in {
    verify(new DecoupledGcd(16),  Seq(BoundedCheck(kMax = 10)))
  }
}
```

If the user clicks the test icon again or runs the `sbt test` command, a formal bounded check will be executed for ten cycles after reset instead of a simulation test. The only program required in addition to the normal Scala development setup is a copy of the open-source SMT solver Z3 [39]. Figure 5.1 illustrates the IDE based workflow.

The check will succeed trivially, no matter which changes we make to the circuit since there are no assertions in the GCD source code. A lack of assertions means there is nothing to tell the solver if the circuit misbehaves. To have something to verify, we can add assertions directly to the circuit by using the Chisel `assert` statement. The decoupled GCD circuit used as an example has an `input` and an `output` channel as well as a 1-bit `busy` register. We expect that while the circuit is busy, no new input is accepted:

```
when(busy) {
  verification.assert(!input.fire())
}
```

This assertion could easily fail in the first execution cycle since the busy register starts in an unknown state when the circuit is powered on. By default, Chisel registers all start in an unknown state and only take on their initial values when the implicit reset wire is asserted for at least one cycle, closely modeling the behavior of actual hardware implementations. However, the assertion actually passes, and the GCD circuit correctly implements the expected behavior. This is because our system always implicitly adds the assumption that the reset pin is asserted during the first cycle of the formal check, similar to how the dynamic testing interface runs the design through a reset cycle before executing the user test. In addition, assertions in Chisel are guarded by the reset signal by default. Thus, the property at hand is only checked after a successful reset, and thus, it holds. Our implementation allows the user to disable the automatic reset behavior for special cases; however, since Chisel typically mandates reset conventions, our automated reset assumptions are what users generally want. We thus simplify formal verification by making the standard case the default while allowing power users to opt out.

Next, we introduce a minor bug into the GCD circuit by connecting `input.ready` to `true.B` and rerun the test. ChiselTest reports an assertion violation one cycle after reset. It also presents the user with an error message indicating the Scala line number of the failing assertion. To debug the problem, they can find a VCD waveform dump in the standard test directory created by our ChiselTest library. Since we replay the test on a concrete simulator, the error message and VCD will be precisely the same as if the user were running a simulation test. Any improvements to our simulation interface or error reporting will thus immediately benefit formal verification users.

A more advanced property we expect to hold is that if the input and output channels are idle, the busy signal will remain the same in the next cycle:

```
when(past(!input.fire() && !output.fire())) {
  verification.assert(stable(busy))
}
```

Here, we use our `past` function for temporal properties, described in Section 5.4. Our system automatically delays the assertion to only be checked one cycle after reset to ensure that past values of the expressions exist.

## Memory Behavior Verification Example

Figure 5.2 shows how we can use ChiselTest with our newly added verification capabilities to verify the read-under-write behavior of memories in the Chisel language. Read-under-write occurs when both the read and the write port of a memory access the same address in the same execution cycle. The Chisel memory primitive can be configured to resolve the conflict in three different ways. The memory can return the old data at the address (`ReadFirst`), the newly written data (`WriteFirst`), or an arbitrary value (`Undefined`). In the example, we encode the `WriteFirst` behavior as a temporal assumption using our new `past` statement. The check fails if `WriteFirst` is substituted with `ReadFirst` or `Undefined` (Section 5.2).

A blogpost [1] with a similar example written in the Verilog language served as inspiration. However, in the Verilog version, the user must manually encode the restriction that the assertion may only be checked after one cycle because otherwise, the `$past` operator in Verilog returns an invalid x value. In the Chisel version, the assertion is automatically delayed until at least one cycle after reset, when there are valid `past` values available (Section 5.4). A bounded model check is executed by the `verify` command, which is called from a standard Scala unit test (Section 5.1). When the check fails, the failing inputs and starting states are replayed on a simulator, resulting in a waveform file identical to the output we would get from a dynamic verification run. However, since we used bounded model checking to find the failing trace, it will be as short as possible. In our example, the design must be executed for two cycles after reset to trigger a failure of the property. The first cycle contains the read

[1] `https://zipcpu.com/answer/2021/07/03/fv-answer15.html`

```
class Quiz15 extends Module {
  /* [...] I/O definitions */
  val mem = SyncReadMem(256, UInt(32.W), WriteFirst)
  when(iWrite) { mem.write(iWAddr, iData) }
  oData := mem.read(iRAddr, iRead)

  when(past(iWrite && iRead &&
            iWAddr === iRAddr)) {
    verification.assert(oData === past(iData))
  }
}

class ZipCpuQuizzes extends AnyFlatSpec
  with ChiselScalatestTester with Formal {
  "Quiz15" should "pass with WriteFirst" in {
    verify(new Quiz15, Seq(BoundedCheck(5)))
  }
}
```

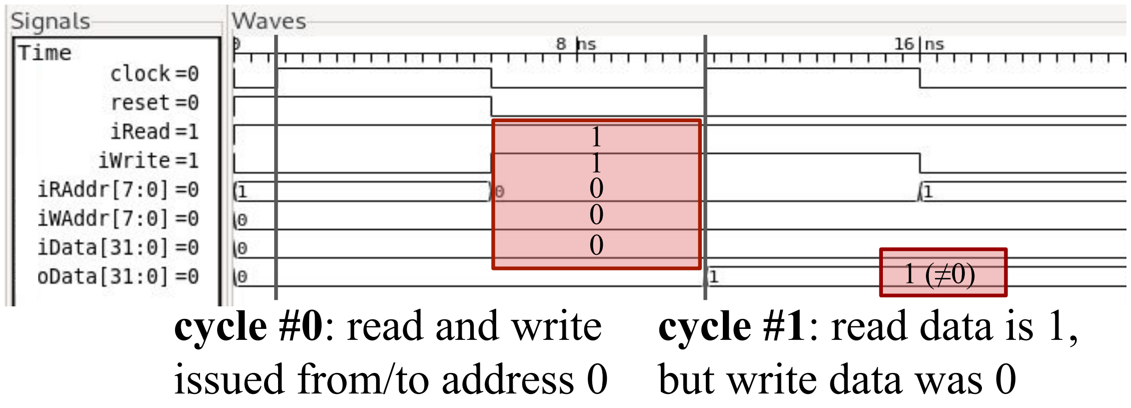


Figure 5.2: ZipCPU verification quiz implemented with ChiselTest.

and write requests. The second cycle observes the arbitrary result on the read port if we set the memory behavior to `Undefined` for read/write conflicts.

## 5.2 A Formal Backend for FIRRTL

The verify command introduced in the previous section elaborates on the Chisel design, translates it into a format that an open-source model checker or SMT solver can understand, invokes that solver, and then communicates the result. In this section, we describe how we pre-process the FIRRTL description of the generated design and finally convert it into the btor2 or SMTLib formats, depending on which backend engine we use. Structural RTL

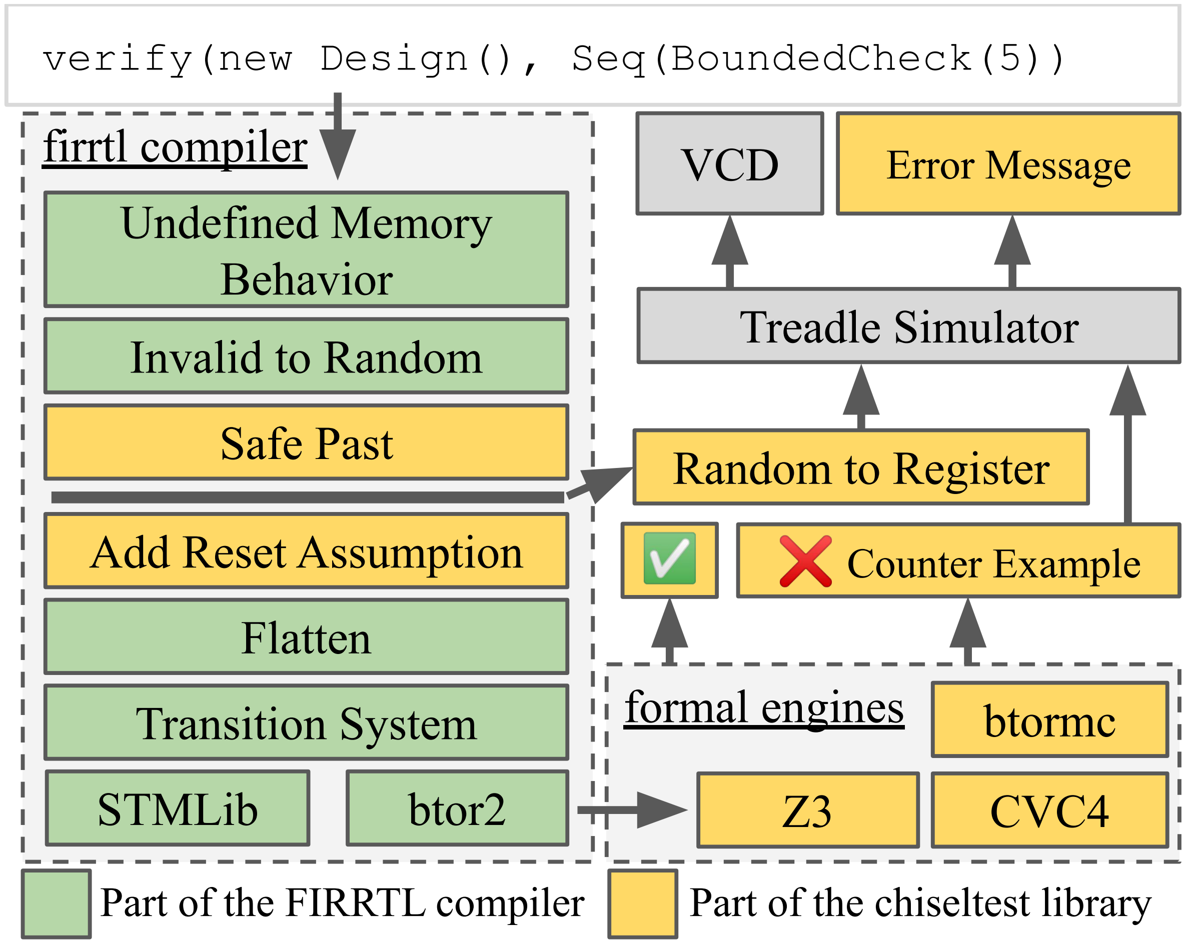


Figure 5.3: The **verify** command is implemented as part of the ChiselTest library and uses several compiler passes that make up the FIRRTL formal backend. We hook into the FIRRTL compiler to model undefined behavior with `DefRandom` statements and to delay temporal assertions as part of our safe past construct. We then add reset assumptions, flatten the system, convert to a formal transition system and then serialize the system to SMTLib or btor2. We provide bindings to launch various formal engines from ChiselTest. If a counter-example is found, we convert the `DefRandom` nodes in the circuit to registers before loading the circuit into the treadle simulator to replay the failure and obtain a simulation quality VCD and error message.

in the LoFIRTL format (see Section 2.1 for more background) is semantically close to the supported backend format and can be translated rather straightforwardly. We did have to add passes to carefully model arbitrary values in FIRRTL, ensuring the model checker can exploit them to find corner-case bugs in the design under test. Finally, we explicitly designed our system to ensure that any counter-example found by a formal backend can be faithfully replayed on our normal simulation infrastructure. We use the treadle interpreter to replay failure-inducing inputs, thus providing the user with high-quality VCD waveforms and easy debugging with Chisel's print statements. Figure 5.3 shows our compilation flow in more detail.

## Output for Model Checkers

Most modern open-source model checkers consume circuits in the simple btor2 format [111]. This format supports no notions of a module hierarchy; all one can express is a circuit that comprises a single module. Thus, our backend flattens the design under test by inlining everything into a single module. To ensure that we produce an excellent waveform dump, the counter-example will be replayed on the non-inlined circuit. We use the FIRRTL compiler's built-in annotation support to automatically track the name changes of all registers and memories in the design as they are inlined. This way, we can map initial states found by the formal engine back to their hierarchical names to accurately initialize the state before replaying a failure-causing input in simulation.

Once the circuit has been flattened, the conversion to a transition system is fairly straightforward. We implemented an SMTLib and btor2 encoding very similar to the one pioneered by yosys. We used the FIRRTL specification to accurately translate FIRRTL expressions to the bit-vector expression language defined by the SMTLib format [8]. Our backend supports memory and registers initialization using the same user annotations as the Verilog backend. Multi-clock support through a clock stuttering pass is a work in progress; for now, only circuits with a single clock domain are officially supported.

## Modelling Non-Deterministic Behaviors

Users want their Chisel designs implemented with as little hardware as possible. The FIRRTL specification was crafted to allow some operations to result in arbitrary results, allowing for better compiler optimizations and avoiding unnecessary hardware. For example, a wire connected to `DontCare` or to the result of a division by zero carries an arbitrary value. Reading from a memory while the read port is disabled, reading from the same address another port is writing to, or writing from two memory ports to the same address all generate an arbitrary value result. The Verilog code generated by the FIRRTL compiler does not reflect all arbitrary behaviors. This is because the compiler is free to substitute arbitrary with (more) concrete values, like always returning a memory read result even when the read port is disabled or assigning a priority to write operations so that at least one will complete. Thus, if we first generate Verilog and then use yosys, we only verify one concrete design translation. Still, there may be other legal translations that would violate the property. Alternative translations might be produced, e.g., in the context of memories, when we use an external SRAM compiler that might try to rely on the fact that write-write collisions can have arbitrary results to generate better hardware. Thus, it is crucial that we carefully model arbitrary values as part of the FIRRTL compiler's new formal backend.

Initially, we implemented arbitrary-value modeling as part of the translation from LoFIRRTL to btor2 or SMTLib output. However, this approach has two downsides: (1) It complicates the backend pass since translation and arbitrary-value modeling need to be handled in the same implementation, and (2) to accurately model arbitrary values in the circuit described by the user code, this modeling has to take place at the beginning of the compiler pipeline, since some existing transformations might otherwise prematurely change or remove behaviors.

Our new approach uses a new `DefRandom` statement to decouple arbitrary-value modeling and emission. It provides a named arbitrary value that can change every clock cycle, much like a `anyseq` annotated wire in Verilog. Our btor2 and SMTLib emission implements `DefRandom` nodes as additional inputs to the design. These inputs are freely controlled by the formal engine, thus allowing any value to be chosen on each step. We implemented a pass that replaces expressions that produce arbitrary values with a value from a new `DefRandom` node. This takes care of explicit assignments with `DontCare` and divisions, which have an undefined result when the divisor is zero. A second pass takes care of modeling four non-deterministic behaviors around memories. (1) when the same address is accessed from a read and a write port in the same cycle, the value on the read port output is arbitrary (2) when a read port is not enabled, the output is arbitrary (3) when the same address is updated from two different write ports, then the address is updated to an arbitrary value (4) when a value is read outside of the memory range, the value on the read port is arbitrary. Out-of-bounds reads can happen because FIRRTL memories can have an arbitrary number of entries. Their depth does not have to be a power of two.

## Interoperability with Simulation

Once the formal engine finds starting states and inputs that lead to an assertion violation, we need to help the user debug their design. Since we do not have the extensive resources of a major EDA vendor, we would like to reuse as much of the existing simulator infrastructure as possible. If we can replay the failing trace on our existing simulator, the VCD waveform dump and the error reporting will be the same quality as when writing a concrete test bench. We carefully designed our arbitrary value modeling – explained in the previous section – to allow for exact replay. While no simulator supports `DefRandom`, we can replace them with registers as part of a compiler pass. We then take the arbitrary values provided by the formal backend and apply them to the registers in the simulator for accurate replay. We also collect metadata to map the names of registers and `DefRandom` nodes in the flattened design back to the hierarchical names used by the simulator.

## 5.3 Reset Assumptions

In Chisel, users rarely need to worry about resets. Registers with reset values are automatically connected to the default reset port of the module, and module instances inherit their reset domain from their parent. In Verilog, users must manually ensure that assertions are only triggered after the circuit has been properly reset. We decided to provide sensible defaults instead. Assertion statements are automatically disabled, closely following the behavior of print and stop statements, which have been part of Chisel since the first release. As part of our formal verification support, we provide a FIRRTL pass that automatically adds a constraint for the rest of the top-level module to be active during the first execution cycle. Thus, by default, users do not have to worry about resetting. Their assumptions will only fire after their circuit has been properly reset, and hence, we ensure that there are no false positives. We provide options for power users to write assertions that are active during reset and to disable reset assumptions or increase the number of reset cycles.

## 5.4 Simple Temporal Assertions

While a simple `assert` statement allows us to specify a property over signals during a single cycle, it is not enough to express properties that require us to reason about multiple cycles. The traditional answer to this problem is temporal assertion languages like SystemVerilog Assertions [62]. However, these are complex to implement efficiently, and no successful open-source implementation has been reported as of now. The community around SymbiYosys has instead advocated for the use of plain assertions with the Verilog `past` function. This function returns the previous value of an expression and thus allows us to write properties that span multiple cycles.

While conceptually simple, the `past` construct, as defined by the Verilog standard, has one major problem: In the first cycle of the circuit execution, there is no past value, and the `past` function always returns X. Thus, the user has to take care to keep track of how many cycles have passed since the verification started and only enable assertions once all past values are valid. This particular pitfall is often the topic of a popular formal verification quiz. In addition to that, `past` values can also be invalid because they happened while the circuit was going through reset.

We made use of some of the unique capabilities offered by Chisel in order to implement what we consider to be a safer version of the `past` function. In the front end, our `past` is a Scala function which creates an appropriate amount of delay registers in the current clock and reset domain. That alone provides functionality similar to the Verilog version of `past`. We go further by annotating the delay register and asking for a FIRRTL pass to be run when lowering the design. This pass looks at a graph of all `past` delay registers and assertions in a module. An edge indicates that the input to the assertion or register is connected to the output of a delay register through combinatorial logic. We traverse the resulting tree (by design there can be no cycles) starting at each assertion to find the longest

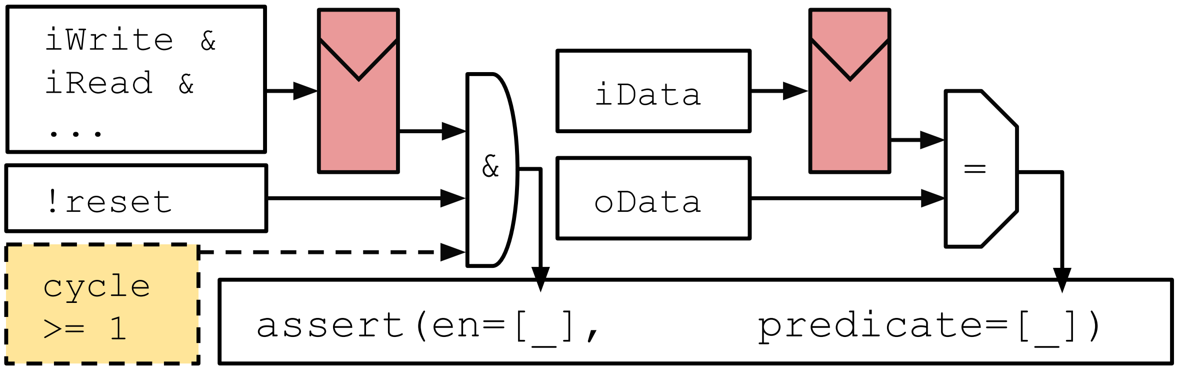


Figure 5.4: The temporal assertion from Figure 5.2 results in a circuit with two registers created by the `past` function: One to delay the condition from the `when` statement and the other to delay the input data before it is compared to the current output data. By default, an assertion is only enabled when reset is inactive and the surrounding `when` condition is true. Our compiler pass analyzes the connectivity graph with the result that both the enable condition as well as the predicate are delayed by a single past register. Thus, the assertion enable signal is automatically extended to include the condition that at least 1 cycle must have passed since the last reset. The new enable condition is derived from a synthesizable, saturating `cycle` counter, which is created by the compiler pass.

path of `past` delay registers in order to determine the number of cycles the assertion needs to be delayed. Finally, we generate a cycle counter register and use its value to guard the individual assertions. Since our `past` function only relies on synthesizable hardware, it can also be used in software and FPGA-based simulation testing [73].

## 5.5 Advantages over SymbiYosys for Chisel

Since Chisel designs can be compiled to SystemVerilog for simulation and physical design, it would be possible to use SymbiYosys to formally verify them. This is, in fact, the approach that some Chisel community members explored after Verilog emission of assert and assume statements was added to Chisel in 2020. However, we encountered multiple problems with that approach, which led us to the implementation presented in this chapter. One problem is that SymbiYosys relies on academic model checkers to generate counter-example waveform dumps for the user to debug. As we pointed out before, academic tools are good at implementing new verification algorithms, but usability features like good Verilog support or good VCD generation are not necessarily their strength. Thus, the generated VCDs are generally of lower quality than the ones our simulator generates. Another problem with SymbiYosys is that it models undefined values in the emitted Verilog code, which differs from those in the original Chisel design. This mismatch can lead to missed bugs as well as false counter-examples.

## 5.6 Discussion

We introduced a new formal verification infrastructure for RTL designs written in Chisel. Everything we described is integrated with our open-source FIRRTL compiler and ChiselTest testing library and thus available to all Chisel users. To lower the barrier to entry, we added default reset assumptions and a safer version of the `past` function for temporal assertions. We carefully designed the formal backend of the FIRRTL compiler to model worst-case behaviors from the FIRRTL specification and to ensure that our system can replay all counter-examples in simulation.

Since we published the formal verification capabilities for ChiselTest in 2021, several community members have started using them in their projects. I have used this functionality to verify a gray code generator that I contributed to the Chisel standard library and for an interactive, Jupyter-notebook-based guest lecture on formal verification for the agile hardware design class at UC Santa Cruz. Unfortunately, the Scala-based FIRRTL compiler has recently been deprecated (see Section 2.4) and replaced with an MLIR-based compiler for the Chisel 6 release. Thus, to make formal verification with ChiselTest work on the latest version of Chisel, we have to port all passes to the new compiler. A Master's student under my supervision recently tackled this task. She managed to upstream a rudimentary btor2 backend and has added support for advanced sequence-based assertions [41]. The new compiler is currently missing the arbitrary-value modeling, reset assumptions pass, simulator replay, safe past operator, and support for FIRRTL memories. I hope this thesis can serve as a blueprint for anybody who wants to work on better formal verification support.

# Chapter 6

# A Study on Random Testing and BMC for RTL Designs

We previously discussed feedback-directed mutational fuzz testing, a form of random testing, in Chapter 4 and bounded model checking (BMC) in Chapter 5. But how do they compare? Is one always better than the other? This chapter defines the three components required for automated bug finding: the input generator, input constraints, and the checker. We discuss how the choice of dynamic or formal verification technology impacts all three and how seamlessly switching between different input generators might be possible.

We present a small study comparing both approaches based on three benchmark sets. We will see how a simple BMC-based baseline can beat the RFUZZ tool on the benchmarks used in Chapter 4 using an idea from Chapter 3. Furthermore, we show how random testing strongly outperforms BMC on a different benchmark set and how BMC can be a significant improvement over dynamic symbolic execution of hardware when used with the correct constraints. The findings in this chapter inspired the RTL-REPAIR tool, which uses a clever combination of dynamic and formal verification techniques and is presented in Chapter 7. All benchmarks discussed in this chapter are available at github.com/ekiwi/comparing-random-testing-and-bmc.

## 6.1 Anatomy of Automated Bug Finding for RTL

Many hardware verification papers, such as a recent paper on security verification [158] and a recent paper on processor testing [71], propose end-to-end solutions for automatically detecting classes of bugs. While these end-to-end scenarios show exciting results and demonstrate the feasibility of the methods in a real-world scenario, they make it hard to compare and contrast different algorithmic approaches. If every paper creates a new verification system from scratch, it is impossible to tell which component contributes to the observed success. Thus, we propose decomposing these verification setups into three core components, which could be mixed and matched to study individual improvements and create more powerful testing systems. The three components are:

1. An *input generator* which produces bug revealing inputs.
2. Declarative or generative *input constraints* ensure the produced execution trace is feasible.
3. One or multiple *checkers* which detect traces that indicate a bug in the circuit.

## Checkers

Checkers are part of a testing setup that detects that a circuit under test malfunctions. They can take on many different shapes depending on their level of sophistication and the verification environment they are used in. In a unittest style setup, the testbench contains hard-coded inputs and expected outputs. The verification engineer has to design scenarios and determine the expected output values manually. With more sophisticated approaches to dynamic verification such as random or constrained random verification *checkers* are often decoupled from the input generator. Monitoring automatons, temporal assertions, or scoreboards monitor the execution of the circuit and raise an alarm if an error is detected. In a recent paper on generating security explits [158], the *checkers* are synthesizable assertions from prior papers [159, 58]. In a recent paper on processor testing [71], bugs are detected by comparing the processor RTL's architectural state against a functional ISA model. Another option is temporal assertions like SystemVerilog Assertions (SVA) [62], which can generally be used with dynamic as well as (commercial) formal verification. Scoreboards and monitors implemented in the non-synthesizable subset of SystemVerilog and often using the UVM library are generally inaccessible to formal verification tools. Tests against a golden software model are also often inaccessible to formal hardware verification tools, which are meant to model-check synthesizable hardware and are incapable of reasoning about arbitrary program code.

## Input Generators

In the simplest setup, the verification engineer hard-codes inputs to the design under test. More sophisticated solutions use pseudo-randomness to generate a stream of different input values. Random testing often requires developers to encode the input constraints into the generator, as discussed in the next section. Instead of generating inputs for concrete simulation of the design under test, we can use formal techniques to generate inputs that satisfy a desired property. Different model-checking and dynamic symbolic execution-based techniques are discussed in Section 2.6. In this chapter, we use the term automated input generator to refer to both formal and random generator techniques.

### Input Constraints

For an automated *input generator* to produce legal stimulus, it needs to be provided with *input constraints* that describe the hardware component's environment assumptions and interaction restrictions. These constraints are often overlooked or only partially stated. They are often heavily tied to the input generator used and thus present the biggest hurdle to combining formal and dynamic techniques.

Dynamic verification environments generally encode complex input constraints in the form of generators, which may, in turn, sample from declarative constraints. A good example is the open-source RISC-V program generator RISCV-DV [122] that can be used to test RISC-V processor implementations. This generator describes a space of RISC-V programs from which we can sample. Unfortunately, model-checking tools generally cannot extract the implicit RISC-V program constraints since they cannot reason about the verification constructs used in RISCV-DV.

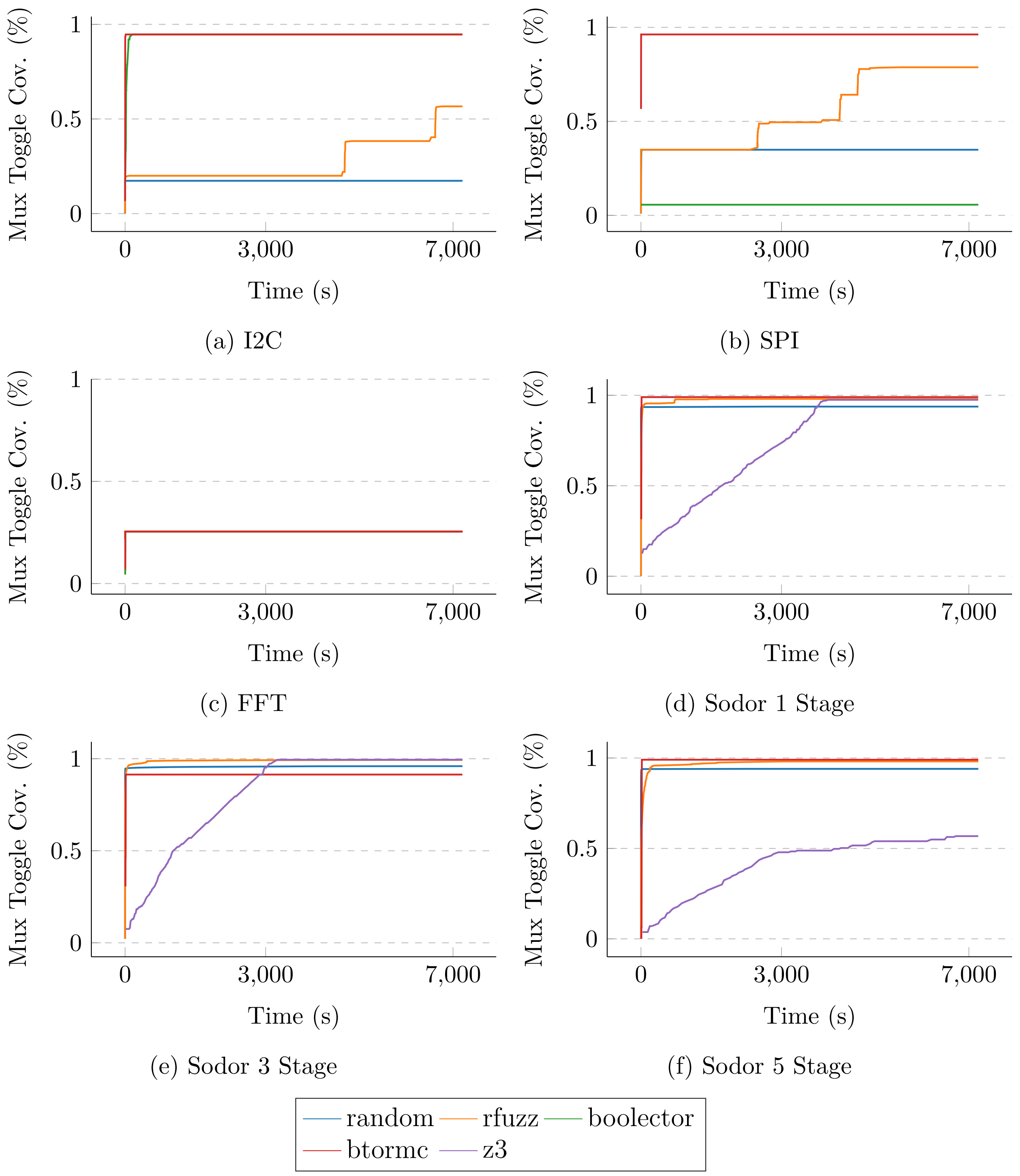
In formal verification, on the other hand, constraints are generally provided as state predicates, which are assumed to hold in every valid execution. These assumptions are easy enough for dynamic tools to consume and check. However, while it is easy to check whether a given input violates an assumption, it can be challenging for random input generators to generate inputs that do not violate complex assumptions efficiently. In the general case, they have to use rejection sampling in which inputs are discarded as soon as an assumption is violated [81]. Rejection sampling can be very slow and inefficient when it is likely that - given a random input - an assumption will be violated.

## 6.2 Coverage Based Fuzz Testing Evaluation

In this section, we compare the performance of coverage-directed fuzz testing and model-checking for hardware.

**Checkers.** Recent hardware fuzzers have been evaluated based on achieving high coverage [81, 141, 24] and on how many bugs they have discovered [60, 72, 22]. Unfortunately, all fuzzers that report bugs are specialized for RISC-V CPUs and use a functional simulator to reveal bugs in the RTL. A formal tool cannot easily analyze the functional simulator; thus, comparing formal and fuzzer in this area is difficult. Coverage is a much easier target for model-checkers since the coverage instrumentation discussed in Chapter 3 uses synthesizable hardware constructs supported by all tools.

We can treat each coverage metric as a set of predicates on the state of the hardware design or the history of states, and thus, hitting a cover point can be formulated as a model-checking problem. This is particularly easy for the benchmarks used in the RFUZZ paper. The tool adds a meta-reset signal, which ensures that all states of the design under test are reset. It makes it fairly straightforward to ensure that the bounded model checking tool does not just reach certain coverage by starting from a cleverly chosen initial state. In



(a) I2C (b) SPI (c) FFT (d) Sodor 1 Stage (e) Sodor 3 Stage (f) Sodor 5 Stage

Figure 6.1: Mux control toggle coverage over time. BMC-based coverage generators generally outperform the fuzzer and the random baseline.

addition, the RFUZZ instrumentation exposes all mux control signals that need to be covered as outputs of the design under test and signals that indicate whether an assertion fires.

To set up the model checking problem for a given RFUZZ benchmark, we create a Verilog test harness that instantiates the instrumented design under test. We then add assumptions that ensure that the meta-reset is active in the first cycle and the real reset pin of the design is active in the second cycle of the execution. For each mux control signal, we add two cover statements that are active after the first two cycles: one to cover the signal being true and one for the signal being false. Thus, we directly encode the coverage metric used by RFUZZ in a way that is usable for bounded model checking. We also add an assumption that ensures that none of the assertions from the original design fire. This is important to avoid generating invalid inputs. The original RFUZZ paper also uses this signal to guide the mutation. See Chapter 4 for more details.

Once we have created the formal harness, we use SymbiYosys [152] in *cover* mode to generate traces that hit one or more coverage statements. Afterward, we parse the generated traces and turn them into JSON files identical to the ones generated by the original RFUZZ tool. We then run the original evaluation script from RFUZZ, which re-runs the fuzzer and BMC-generated inputs on the same RTL simulator to measure the coverage achieved. This ensures a fair comparison.

We find that BMC beats RFUZZ on all benchmarks from the original paper, often with a good margin. However, there is some variability with the different BMC tools. Boolector [110], for example, ran out of memory on all the Sodor benchmarks, and we had to switch the btormc. In some cases, one model-checking tool generated better coverage than another, even though both explored all possible inputs up to the same bound. It is unclear whether this is due to simulation mismatch, bugs in the encoding, or the tools themselves. In theory, all BMC tools should be able to get identical coverage for a given depth $k$. The best BMC tool was btormc for all benchmarks besides Sodor 3 Stage where it quickly finished a run up to $k = 100$ but without reaching maximum coverage. On four benchmarks, btormc always finished in less than 20 seconds and generated better coverage than the fuzzer. On the FFT benchmark, all tools quickly get stuck, indicating that most coverpoints in that design are most likely unreachable. Figure 6.1 shows coverage plots over time, combining the original RFUZZ results from Chapter 4 with our BMC-based comparison. Table 6.1 shows the number of coverage holes and the time it takes to achieve maximum coverage for the different approaches.

This comparison shows that the benchmarks we used to evaluate RFUZZ are unsuitable for highlighting the strengths of fuzzing tools since most of them are easily beaten by the btormc model checker. Going beyond this particular set of benchmarks, our results also indicate that BMC should be used for all automated testing papers to establish a baseline to beat. This is especially true when working with small to medium size RTL designs.

| Design | random | rfuzz | boolector | btormc | z3 |
|---:|---:|---:|---:|---:|---:|
| I2C | 62 (1s) | 5-60 (80min) | 4 (151s) | 4 (14s) | |
| SPI | 69 (8s) | 7-69 (45min) | 100 (3) | 4 (4s) | |
| FFT | 85 ($< 1s$) | 85 ($< 1s$) | 85 (6s) | 85 (4s) | |
| Sodor 1 | 11-13 (3s) | 4 (12min) | OOM | 2 (14s) | 5 ( 67min) |
| Sodor 3 | 6-8 (68min) | 0-2 (46min) | OOM | 16 (15s) | 1 ( 54min) |
| Sodor 5 | 12-13 (38min) | 4 (35min) | OOM | 2 (20s) | 92 ( 112min) |

Table 6.1: Number of coverage holes and best time to achieve maximum coverage across rfuzz benchmark designs and input generators. Note that for fuzzing, we do not consider the time it takes to compile the simulation of the design under test, while BMC times include the time it takes to parse and load the design.

## 6.3 OR1200 Processor Security Bugs

In the previous section on fuzz testing, we could not study CPU fuzzers because they used a complex functional simulator to check for bugs. While we could not find any CPU fuzzing papers that we could easily use with a model-checker, we found a different line of work that uses simple hardware assertions to discover security bugs in the open-source OR1200 CPU [58, 159, 158]. The initial papers from 2015 and 2017 mostly rely on hand-crafted exploits to test the assertions. However, the final paper proposes using a dynamic symbolic execution-based approach to generate instructions that expose bugs in the processor design by violating one of the provided security assertions [158]. In this section, we present the results of a small case study using bounded-model checking instead of symbolic execution to discover the bugs in the original paper.

We use the OR1200 CPU core combined with the assertions and bug implementations found in the paper artifact available online [1]. Unfortunately, the artifact only includes these for two of the 31 bugs discussed in the paper. We emailed the authors asking for the other assertions used in the paper but did not receive a reply. Thus, we can only discuss bugs 20 and 24 from the original paper.

The original paper compares their symbolic execution-based input generator to a commercial model checker from Cadence and the EBMC model checker. However, the authors report that while these tools find many bugs, often the generated inputs do not allow them to trigger the bug on a simulated processor, i.e., they are not repeatable. This includes the two bugs we are looking at. While the paper lacks an artifact that would allow us to repeat the exact experiment, our own experimentation leads us to the following theory: We often require a specific constant in a particular processor register to trigger a security assertion. By default, the Verilog implementation of the register file leaves the initial values unconstrained. Since BMC will always find the shortest execution to trigger a bug, the model

[1] `https://github.com/rzhang2285/Coppelia`

checkers choose a specific initialization of the register file that contains the correct magic number. However, on a real processor, we cannot control the initialization of the register file. Instead, we must execute additional instructions to initialize the register file properly.

The problem with correctly initializing the register file demonstrates an issue with traditional formal verification systems: Since they were designed to prove the absence of bugs, it is enough to show that an initial state exists that allows them to trigger the bug. However, showing that such an initial state exists is not enough to be able to replay the bug on real hardware where we do not control the initial values of registers and memories. Thus, to obtain repeatable exploits, we manually instrumented the processor design such that all memories and registers contain zero in the initial state. Thus, we force the model checker to generate instructions that synthesize any non-zero constants needed to trigger a particular bug. We also modeled an uninitialized instruction memory and disabled access to the data memory to force the model checker to generate a self-contained instruction stream that developers can easily run on a real processor. The only caveat of our approach is that one would have to supply some instructions to manually initialize all registers to zero. However, the original papers also rely on additional instructions to make their exploits feasible.

With our setup, we found that a bounded model checker based on yosys [153] and the boolector SMT solver [110] can generate a counterexample for each bug in under five seconds, including two seconds of actual solving time. While the original paper did not include the time to find bug 20 with dynamic symbolic execution, it does state that after several optimizations, the fastest time to solve bug 24 was 2m33s. Thus, BMC is significantly faster than the new symbolic execution technique proposed by the authors. We also found the restrictions we imposed to generate repeatable exploits to work well. The original paper reports that the Cadence model checker only generates a single instruction: `l.addi r0, r1, 0`. This instruction requires the registers r0 and r1 to be initialized to a particular constant. Our solution, on the other hand, generates an additional preceding `l.movhi r1, 0x409` instruction, ensuring that r0 is non-zero.

Overall, the two security bugs for which buggy RTL code and security assertion are available can be solved quickly by a standard open-source BMC tool. The most difficult challenge is to design a test bench that ensures repeatable results. If we want more engineers used to dynamic verification to take advantage of BMC, we need them to have the option to produce witnesses independent of the starting state. Our findings demonstrate why comparing different *input generators* is important, but also how incompatible *input constraints* can make a comparison difficult.

## 6.4 Deepbugs

We observed bounded model checking to excel at the RFUZZ and the security benchmarks discussed in the previous sections. However, are there any benchmarks where random testing has an advantage over formal approaches? A recent paper investigates partial order reduction

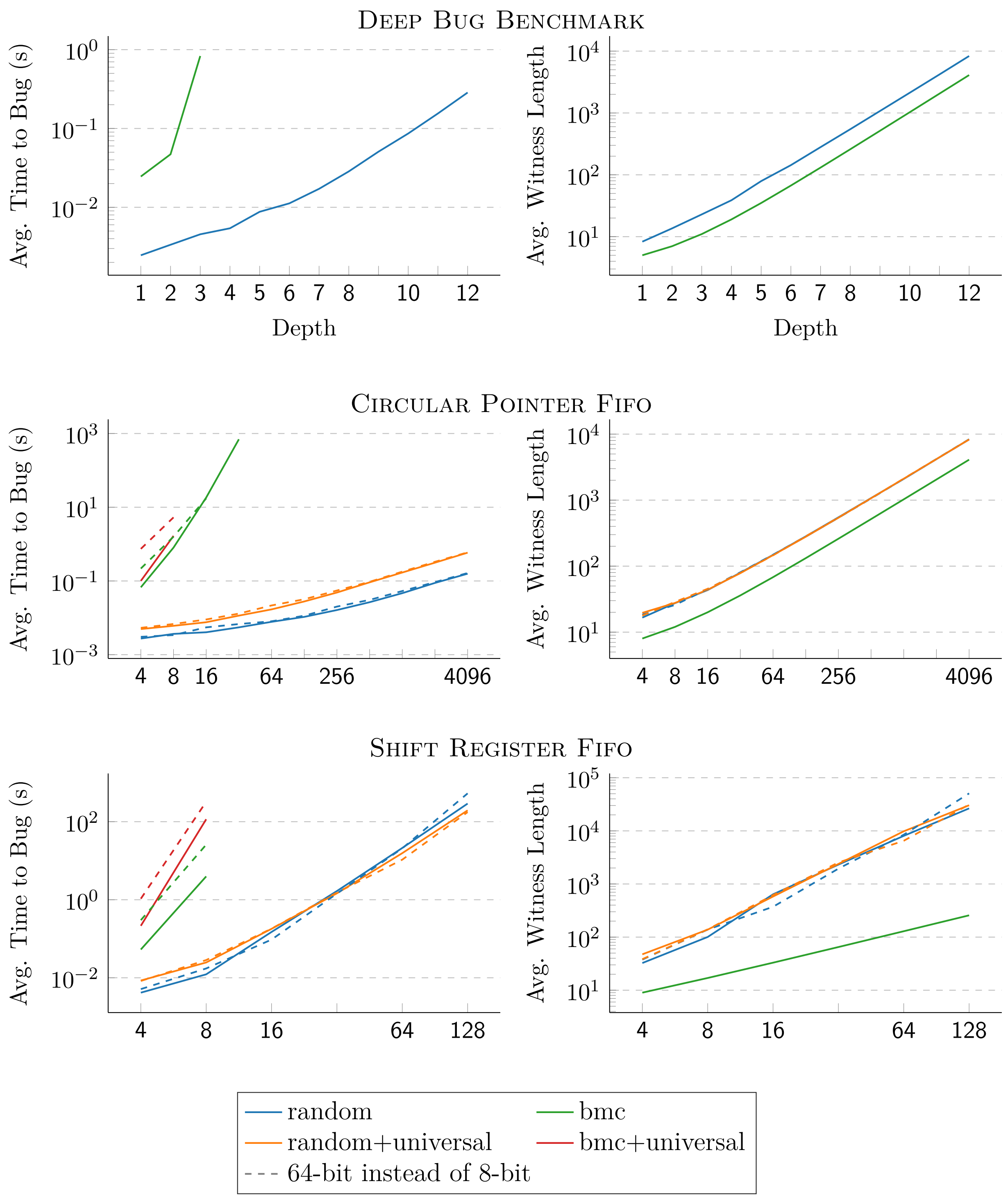


Figure 6.2: Average time to find a bug and average length of the witness over the depth of the design.

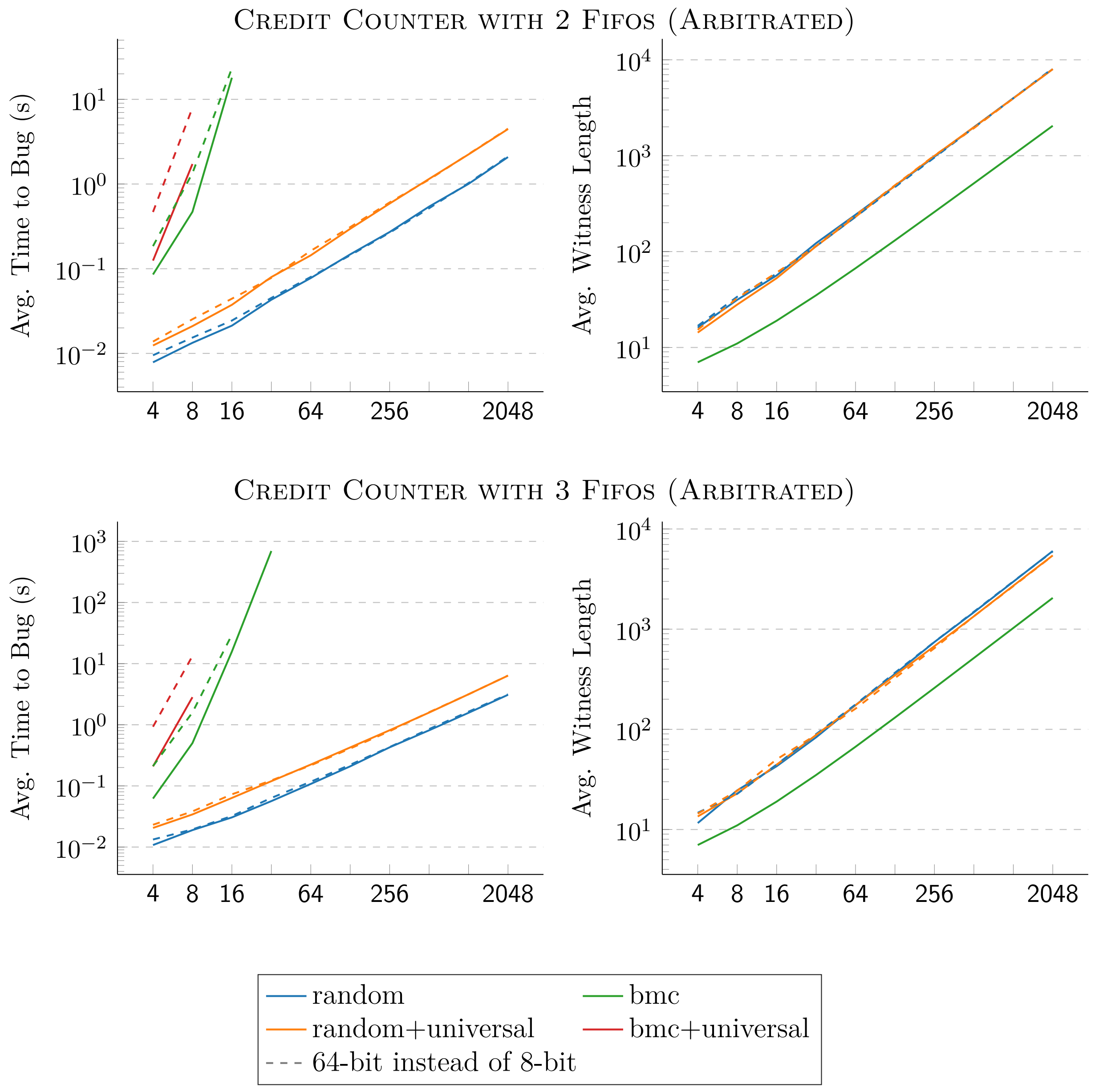


Figure 6.3: Average time to find a bug and average length of the witness over the depth of the design on a second set of benchmarks.

to improve model checking performance for some packet mover benchmarks, which seem to take an exceedingly long time to solve with SAT and SMT-based BMC tools [95].

We ported all of the open-source benchmarks described in the paper to the Chisel hardware construction language [7], allowing for easy parameterization and creation of various test harnesses. Our random tester is based on the Treadle interpreter for the FIRRTL intermediate representation [67], which we introduced in Section 2.5.

Our random testing relies on two different kinds of software harnesses: One is a design-agnostic harness that applies a new random value to each input pin on every cycle. This is similar to how inputs are provided to designs under test by the RFUZZ fuzzer described in Chapter 4. While this harness is very flexible, it does not know about any *input constraints* and does not include a *checker*. Thus, it has to rely on the design under test or an additional hardware harness to encode constraints and assertions. This is similar to how we encode these constraints for bounded model checkers. The second kind of software harness is specific to the device under test. We created one for the FIFOs and one for the arbitrated design, which ensures that push and pop commands are only applied if they won't violate an assumption of the design under test. These software harnesses also contain instances of the Queue class from the Scala standard library, which serve as software models for the FIFOs in the hardware design and allow us to check dequeued values in software and raise an exception if a bug is detected. This harness design is close to how software testbenches are normally written. However, it is incompatible with formal approaches.

The random tester ensures that the reset pin of the design under test is activated for the first cycle of execution and remains de-asserted after that, no matter which version of the harness we use. The tester keeps executing the software harness until an exception is thrown either by software or through an assertion failure in the hardware under test. We rerun all random testing 10 times and report the average time to find a bug and the average length of the counter-example found to account for the stochastic nature of the tool.

The Deep Bug design, which serves as an artificial example in the original paper, has no *input constraints* and contains a single assertion indicating that the bug was detected. Thus, it can be directly used with a bounded model checker or our random tester using the design-agnostic software harness described in the previous section.

The next benchmark features two FIFOs with no assertions or assumptions. Instead, the original paper relies on a *magic packet tracker* that remembers the value of a packet that enters the FIFO in an arbitrary cycle and counts the number of packets exiting the FIFO until the tracked data is expected. This circuit relies on a non-deterministic decision about which packet to track. This is sufficient for formal methods but not very suitable for random testing. If we randomly choose whether to track a packet or not, it makes it much less likely that we detect a bug, even if all other design inputs are chosen correctly. Thus, we track all packets going through the FIFO using a Scala Queue in our specialized software harness. We also add some logic to all harnesses to ensure that no push happens when the FIFO is full and that no pop happens when the FIFO is empty. Using a golden model, like our Scala Queue implementation, is standard practice for software testbenches but makes them unusable for formal tools.

In addition to the software-only harness, we also designed a universal harness that can be used with both the random and the bounded model checker. To do so, we essentially implement the exhaustive tracking of all FIFO packets in hardware through a trusted reference queue implementation from the Chisel standard library. While this universal harness revealed the bug with about the same counterexample size as the software harness, it generally proved slower for both random testing and BMC. We tested FIFOs with an 8-bit and a 64-bit entry size, but the difference in runtime and counter-example length was mostly negligible. While the universal approach is slightly slower, the fact that it is *input generator* independent could save a lot of developer time.

The final benchmark is an arbitrated credit counter circuit, which instantiates several FIFOs and credit counters. It has the most complicated *input constraints* of any benchmark. We implemented a software-only harness using several Scala Queues as golden reference models and a universal harness that implements a simple hardware reference model based on the queue implementation from the Chisel standard library. Both harnesses also contain logic to encode the input constraints to avoid driving illegal inputs to the design under test.

The results of our benchmarks are shown in Figure 6.2 and in Figure 6.3. Random testing was much faster than BMC, which often timed out after 15 minutes. We manually calculated the minimum size of the counter-example for all depths. In general, random testing resulted in a counter-example (or witness) around two times as long as the minimum counter-example BMC would find. The only exception is the shift register FIFO benchmark, for which random testing is timing out after a depth of 128 and generally produces longer counterexamples. This indicates that hitting this particular bug is less likely than in the other benchmarks.

Our results show that simple random testing can outperform BMC significantly, even on small RTL components like hardware queues. Some manual work was required to design a software harness for each benchmark, and counterexamples are longer than needed, likely making it harder to debug them. If these two problems could be addressed, random testing might be competitive with BMC tools at solving buggy benchmarks, for example, at the hardware model checking competition [118].

## 6.5 Discussion

In this chapter, we investigate bug-finding benchmarks from three prior works with surprising results: The coverage over time metric used to evaluate some fuzzing tools can often be beaten quite easily with a bounded model checker. Processor security exploits for which a prior paper developed a new dynamic symbolic execution-based input generator can quickly be synthesized by bounded model checking with easily replayable results as long as the test harness is designed with that in mind. A set of packet mover benchmarks that are hard to solve with BMC are actually quite easy to solve – in many cases – by random testing, with the only major downside being the increased counter-example length. Overall, we show that both techniques are often complementary, and users might benefit from being able to quickly switch the *input generator* that they are using.

Our experiments show that bounded model checking is an important baseline when verifying small to medium-size RTL designs from FIFOs to CPU cores. While formal verification tools that work directly with standard hardware languages like Verilog [153] and Chisel [78] are now widely available, many pitfalls remain. We thus recommend following our test harness approach, which prefers soundness, i.e., the ability to easily replay every bug over completeness by initializing all registers and memories to zero. Using generators instead of declarative *input constraints* and *checkers* that do not rely on non-deterministic decisions, it is possible to create test harnesses that work for both BMC and random testing.

Other benchmarks are challenging for BMC, but random testing can quickly find assertion violations. Unfortunately, we had to manually re-write testbenches originally designed for BMC to make them work well with random testing. This effort prevents developers from easily experimenting with different *input generators*. Suppose we could develop an automated transformation for this process and combine it with automated test case reduction to counter the fact that random witnesses are unnecessarily long. In that case, one might be able to develop a tool that vastly outperforms BMC for finding bugs on specific benchmarks.

Conversely, constraint random testbenches written for dynamic verification could benefit from a model checker-driven exploration. The main challenge would be to soundly turn the information in a software testbench into a formula for an SMT solver. Dynamic symbolic execution of the testbench where every random decision is represented with a new auxiliary variable could be a viable candidate. If the testbench is written in a language with good bounded model checking support, hardware and testbench model could also be combined directly [32].

Our findings show that neither BMC nor random or fuzz testing can solve all benchmarks perfectly. We look forward to new ideas on combining them, which can be incorporated into new testing languages, tools, and methodologies.

# Chapter 7

# Fast Symbolic Repair of Hardware Design Code

So far we have presented new approaches to automatically generate inputs for our design under test through fuzzing (Chapter 4) and bounded model checking (Chapter 5). However, only discovering bugs by finding a failing test input is not enough. An engineer still needs to analyze the failing trace to determine how to fix the problem. In this Chapter, we present an automated tool called RTL-Repair, which takes in a circuit written in the Verilog hardware description language as well as a failing test bench in the form of a trace of inputs to the design under test as well as executed outputs. From this information, RTL-Repair tries to generate a change to the original Verilog that will make the repaired circuit pass the I/O trace. Our work on RTL-Repair was published as ASPLOS in 2024 [79].

The failing I/O trace could be obtained from a manually written test, from fuzzing, or from an SMT solver, which produced a counter-example for a property using bounded model checking. The classic way of debugging this failing trace is for the designer to look at a rendering of it and use their knowledge of the design to try and find a way to fix it. Some recent academic work has also looked into source-level debugging for hardware languages [157], but traditional debugging tools from software, such as step-through debugging, are not always as useful in the hardware context. While software programmers are used to thinking of their programs as executing strictly in program order, multi-threaded programs break this abstraction, which makes them much more challenging to reason about. In hardware, there

Table 7.1: RTL-Repair vs state-of-the-art tool.

| | RTL-Repair | | | CirFix [2] | | |
|---|---|---|---|---|---|---|
| | # | median | max | # | median | max |
| ✔ Correct Repairs | 16 | 0.70s | 13.17s | 10 | 2.53min | 14.19h |
| ✖ Wrong Repairs | 2 | 0.51s | 0.68s | 11 | 2.03h | 9.50h |
| ○ Cannot Repair | 14 | 5.64s | 59.81s | 11 | 16.00h | 16.00h |

are no sequential programs. Results and state updates are all computed concurrently. HDLs reflect that by modeling components as a composition of parallel processes (in Verilog [62] and VHDL [64]) or by allowing signals to be used before they are assigned (last-connect semantics in Chisel [7]). RTL-Repair sidesteps this problem by directly suggesting relevant changes to the RTL developer.

The software engineering community has long been working on automated program repair [83, 55, 100, 89]. In the standard scenario, we are provided with program source code and test cases, at least one of which currently fails. The tool then tries to find one or several changes to the source code, which makes all test cases pass. Unfortunately, most automated program repair tools take several hours to run and often provide unsatisfying repairs, which remove program functionality [119]. Recent work on a tool called CirFix shows that automated program repair can be applied to hardware descriptions as well [2, 126]. However, CirFix can take several hours to come up with a repair and often results in unsatisfactory repairs.

In this paper, we present RTL-Repair, which produces more correct repairs than CirFix in a fraction of the time (Table 7.1). We demonstrate how to combine the repair template idea from CirFix with symbolic analysis-based repair and how to address scalability issues associated with long-running testbenches. RTL-Repair is available on github: `https://github.com/ekiwi/rtl-repair`. We also provide an artifact with scripts to reproduce all our results. Our paper makes the following contributions:

- We propose a new symbolic, template-based repair algorithm
- We introduce an adaptive windowing technique that allows us to scale to long-running testbenches
- We define a new output/state divergence delta (OSDD) metric that helps reason about the hardness of bugs
- We perform a thorough evaluation of RTL-Repair and CirFix, including gate-level simulation as a new way to automatically verify repairs of hardware
- We further evaluate RTL-Repair on real bugs mined from open-source projects [90]

## 7.1 Repair Example

To illustrate key components of the RTL-Repair algorithm, we present an example before diving into details in Section 7.2. We are going to repair the Verilog description of a simple counter circuit (Figure 7.1a). This is the same example circuit that was used in the CirFix paper [2]. We first illustrate how the circuit can be converted to perform BMC with an SMT-solver before showing how RTL-Repair adapts BMC for its repair algorithm.

```
module first_counter (
   input clock, input reset, input enable,
   output reg [3:0] count,
   output reg overflow
);
always@(posedge clock) begin
 if(reset == 1'b1) begin
   // count reset is missing:
   // count <= 4'b0;
   overflow <= 1'b0;
 end else if (enable == 1'b1) begin
   count <= count + 1;
 end
 if(count == 4'b1111) begin
   overflow <= 1'b1;
 end
end
endmodule
```

(a) Verilog source code.

```
overflow' =
 (ite (= count (_ bv15 4)) true
 (ite reset false overflow))
count' =
 (ite reset count
 (ite enable (bvadd count (_ bv1 32))
```

(b) Next state expressions in SMTLib format.

Figure 7.1: A counter circuit with a missing reset value.

**Transition System Encoding.** Before we can formally analyze the circuit, we need to convert the Verilog code into a format that is amenable to formal analysis. We use the open-source synthesis tool yosys [153] to turn the event-driven simulation into a circuit-like transition system representation that encodes the clock updates for the `count` and `overflow` registers as SMTLib [8] bit-vector expressions (Figure 7.1b).

**I/O Trace.** The bug we are looking to fix is revealed by a simple test: After we reset the circuit, we expect the `count` output to be zero. However, currently, it is X since the `count` register is missing a reset assignment. RTL-Repair accepts concrete tests in the form of I/O traces – essentially tables with one row for every execution cycle and one column for every input and expected output value. Figure 7.2a shows the trace for our small example test. Besides manual entry, an I/O trace can be recorded from a concrete testbench, similar to how CirFix obtains expected outputs for its fitness function. It could also be returned by a BMC tool that has discovered a bug in the circuit. We designate inputs that could be set to any value with $X$. For outputs, an $X$ indicates that the value of the output at that particular time step does not matter, i.e., it is not checked by the testbench.

**Repair Template.** RTL-Repair analyses the Verilog source code and enumerates all possible changes to the circuit that fit a certain template. In our example, we consider assigning a constant to a signal somewhere in code. For each assignment, we create two new inputs: $\phi_i$ and $\alpha_i$. $\alpha_i$ represents a constant that can be freely chosen by the repair synthesizer. $\phi_i$ indicates whether the assignment should be included. Figure 7.2b shows how we add two possible new assignments to the circuit. Generally, we will add a lot more possible assignments. However, we restrict ourselves to two in this example to make the resulting synthesis query easy to understand. For a more thorough description of the various repair templates, please see Section 7.2. The instrumented Verilog AST is converted into a transition system using yosys [153].

**The Basic Repair Synthesizer.** RTL-Repair unrolls the transition system exactly as we would for bounded model checking. However, instead of asking the solver to choose the inputs, we assert that the input and output values are equal to the ones from the given I/O trace and ask the solver to provide an assignment to our synthesis variables $\phi_i$ and $\alpha_i$ such that the circuit correctly follows the I/O trace. Such a repair query is shown in Figure 7.2c. Figure 7.2d shows two solutions found by the solver. Both solutions add an assignment to the reset block. The difference is that the second solution also adds an assignment in the overflow code block. The I/O trace never increments the counter all the way to 15, which makes this new statement dead code in terms of the test we provided. However, assigning `count` to 0 in the overflow block, as the solver suggests, introduces a new bug in our circuit, which is revealed if we test the overflow behavior. We generally find that the fewer changes we make, the more likely we will arrive at a valid repair. We thus implement an algorithm that ensures a repair with a minimal number of changes.

| $reset$ | $enable$ | $count$ | $overflow$ |
|---|---|---|---|
| 1 | $X$ | $X$ | $X$ |
| 0 | 0 | 0 | $X$ |

(a) I/O Trace Generated from a Testbench.

```
module first counter (
   // [...] I/O from original circuit
   input φ0, input [3:0] α0,
   input φ1, input [3:0] α1);
always@(posedge clock) begin
 if(reset == 1'b1) begin
   overflow <= 1'b0;
   if(φ0) count <= α0;
 end else if(enable == 1'b1) begin
   count <= count + 1;
 end
 if(count == 4'b1111) begin
   overflow <= 1'b1;
   if(φ1) count <= α1;
 end
end
endmodule
```

(b) Simplified conditional overwrite template applied.

```
; random concrete initial state
(assert (= overflow@0 true))
(assert (= count@0 (_ bv8 4)))
; next state
(define-fun count@1 () ( BitVec 4)
 (ite (and (= count@0 ( bv15 4)) φ1) α1
 (ite (and reset@0 φ0) α0
 (ite (and (not reset@0) enable@0)
   (bvadd count@0 (_ bv1 4)) count@0))))
; I/O trace
(assert reset@0)
(assert (= count@1 ( bv0 4)))
; limit number of changes to one
(assert (= #b01 (bvadd
  (ite phi1 #b01 #b00)
  (ite phi0 #b01 #b00))))
```

(c) Repair query.

| $\phi_0$ | $\alpha_0$ | $\phi_1$ | $\alpha_1$ | change size: $\sum_i^{N=2} \phi_i$ |
|---|---|---|---|---|
| 1 | 0 | 0 | $X$ | 1 |
| 1 | 0 | 1 | 0 | 2 |

(d) Possible solutions.

Figure 7.2: Repairing the counter circuit from Figure 7.1

First, we use the solver to check whether a solution with any number of changes exists at all. If that is the case, we then search for a solution with a minimal number of changes by calculating $\sum_i^N \phi_i$ in the SMT query and successively increasing the number of changes we want to see until the solver returns a satisfying assignment. This constraint is demonstrated at the bottom of Figure 7.2c. By restricting the number of changes to one, we obtain the minimal solution with $\phi_0 = 1$ and $\phi_1 = 0$. While simple, the major downside of the basic repair synthesis approach is that we always unroll the system for all cycles in the I/O trace. This leads to scalability issues with long testbenches, which can be solved by our new adaptive windowing approach described in Section 7.2.

**Repairing the Verilog Code.** We use the repair synthesizer's assignment to the synthesis variables to generate the repaired Verilog code. We remove any assignment where $\phi_i$ is false. This can be thought of as plugging in the assignment from the synthesizer and running a simple dead-code elimination. We inline the concrete value for $\alpha_i$ for all remaining code, where $\phi_i$ is true, and thus, the assignment happens unconditionally. After making these changes on the AST, we serialize it into a repaired Verilog file.

## 7.2 The RTL-Repair Repair Algorithm

The RTL-Repair tool accepts a buggy Verilog module and an I/O trace as input. It first runs a standard static analysis tool to address some straightforward errors that would lead to non-synthesizable code. Next, RTL-Repair applies a series of repair templates, each implemented as a compiler pass over the Verilog AST, which adds different ways for the repair synthesizer to fix the circuits. Our repair synthesizer takes the transition system (converted from Verilog with yosys) and the testbench in the form of an I/O trace as input. It then tries to find a minimal change from the space of changes described by the repair template that will make the circuit pass the test. If such a minimal change is found, it is applied to the Verilog AST, resulting in a repaired source code. If the change is large ($\sum_i^N \phi_i > 3$), then we keep on trying out templates to see if a smaller repair can be found with a different template. If no change can make the I/O trace pass, RTL-Repair will move on to the next repair template. Once all repair templates have been tried with no success, the user is notified that no repair could be found. The whole process is illustrated in Figure 7.3.

### Preprocessing with Static Analysis

RTL-Repair's symbolic repair algorithm requires the buggy design to be synthesizable [61, 135]. This is generally not a problem. In industry, static analysis tools called linters enforce coding standards that guarantee that the circuit can be synthesized. Modern hardware languages like Chisel allow users only to express synthesizable circuits [7]. A study of bugs in

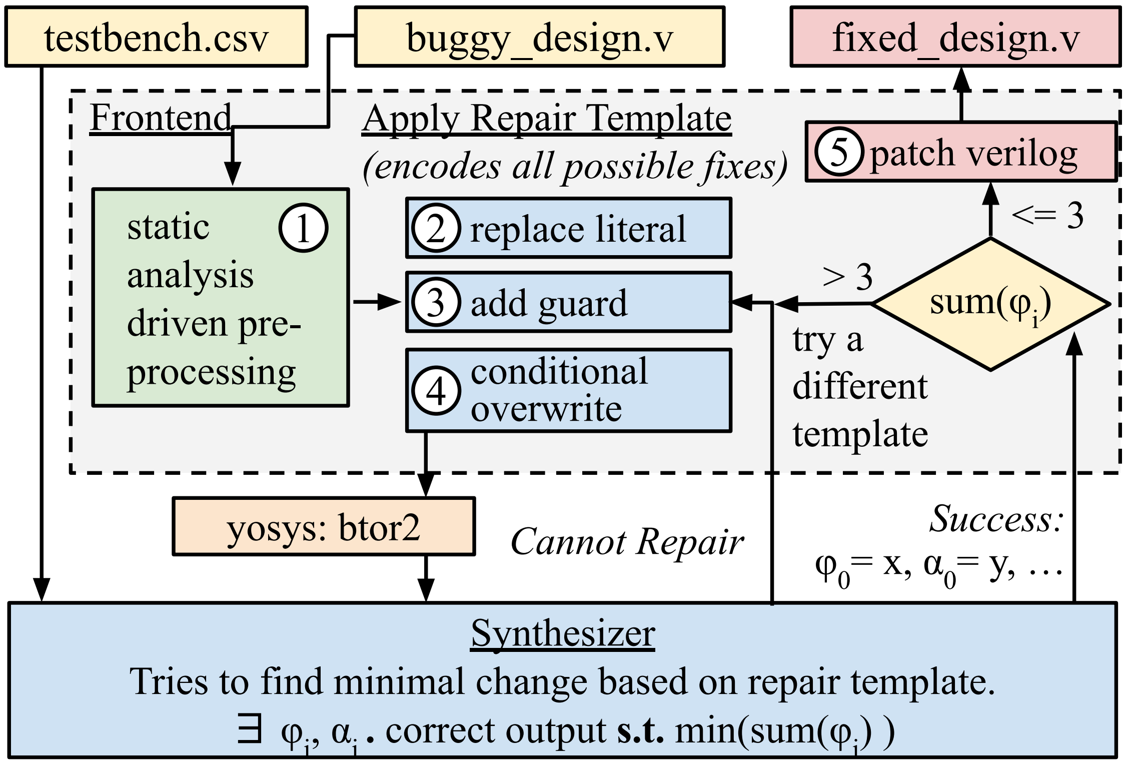


Figure 7.3: RTL-Repair Flow

open-source hardware projects found no issues with synthesizability in practice [90]. However, novices might still make these kinds of mistakes in Verilog, which are pervasive in the benchmarks targeted by CirFix [2]. Thus, we employ the open-source Verilog simulator Verilator as a linter [131] to deal with two common issues preventing a circuit from synthesizing.

**Blocking and Non-Blocking Assignments.** Synthesizable Verilog for synchronous circuits generally consists of two different kinds of processes: Ones that describe combinational logic, marked by a sense list that triggers re-computation on the change of any signal, and processes triggered by clock events that describe synchronous logic (registers and memories). By convention, combinational processes use blocking assignments, and synchronous logic processes use non-blocking assignments [37]. If the linter warns about the wrong kind of assignment, we automatically change it to the appropriate version depending on the type of process to ensure correct synthesis with yosys [153].

**Latches** are state elements that get updated when their input changes. In modern ASIC technologies, latches are generally disallowed in favor of clock edge-triggered flip-flops. Latches also cannot be represented in the transition system format used by RTL-Repair. Many

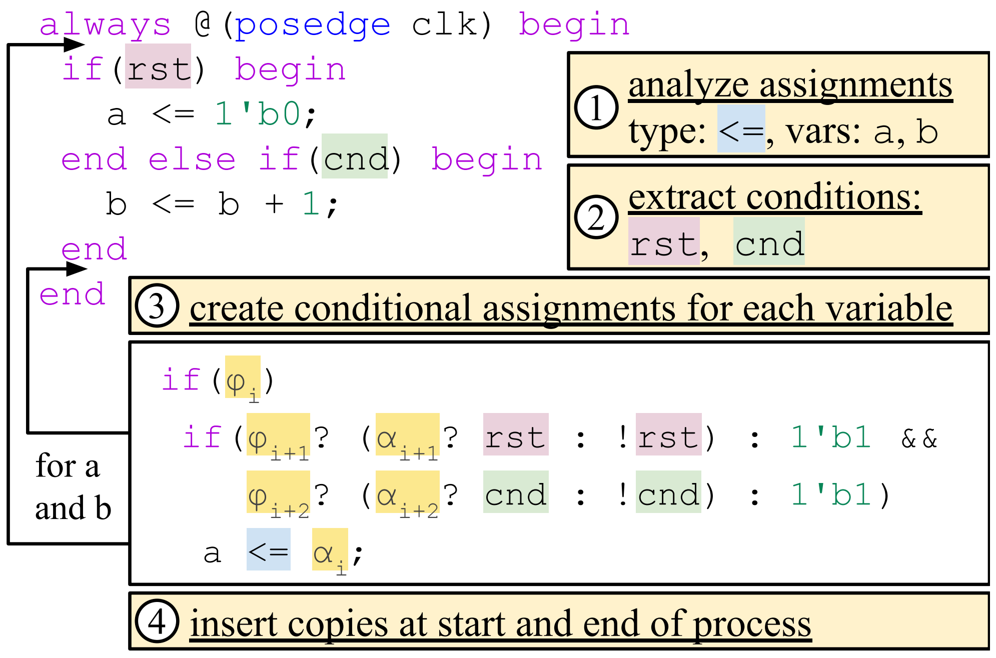
Verilog beginners will unintentionally write code that describes a latch instead of the combinational logic they intended to encode because of a missing assignment in a process. We remove any latches in the Verilog description by providing a default value for a signal whenever there is a warning from the static analysis tool about a latch. We use zero as a default value since it is always valid to assign regardless of the bit-width of the signal. If needed for a repair, the default can be overwritten by the Replace Literals repair template introduced in the next section.

## Repair Templates

A repair template is a compiler pass that analyzes the Verilog AST and adds a range of possible changes, thus describing a space of possible repairs for the repair synthesizer. Each change is guarded by an indicator variable $\phi_i$, which will disable the change when set to zero. Besides that, many templates introduce additional free variables $\alpha_i$, which represent constants in the Verilog code that the repair synthesizer can freely choose. If $\sum_i^N \phi_i = 0$, we turn off all changes and obtain the original circuit. The repair synthesizer will try to find an assignment to all synthesis variables $\phi_i$ and $\alpha_i$ that makes the circuit obey the I/O trace subject to $min \sum_i^N \phi_i$. Minimizing the number of changes has two advantages: (1) it ensures that the synthesizer does not change code that is not relevant to the given I/O trace, making it more likely that the fix will generalize to other tests (2) the smaller the suggested repair, the easier it will be for a developer to verify. We have developed three different repair templates that can fix a wide range of bugs. In our framework, new repair templates can be easily added without any changes to the repair synthesizer as long as they use $\phi_i$ and $\alpha_i$ variables as described above.

Our symbolic repair templates are inspired by the templates used by CirFix [2]. However, while applying a CirFix template will produce a single concrete change to the RTL description, applying a RTL-Repair template encodes a large set of changes that the synthesizer can choose from. Concretely, while CirFix's *Conditional* repair template will pick a single random conditional to invert, our *Add Guard* template will present the synthesizer with the possibility to invert every single condition in the RTL description. Our templates are thus much more powerful compared to CirFix's, which explains why three templates are enough for RTL-Repair to solve many benchmarks.

**Replace Literals Template.** This template allows the repair synthesizer to replace literal integer values with a freely chosen constant. To ensure that we obtain a synthesizable circuit, we restrict the integer literals that can be replaced with the ones appearing in r-value expressions. Thus, we exclude integer literals that specify signal types (bit-width), parameters, and any other integer literals that cannot be replaced with a non-constant expression. Figure 7.6 shows some examples of integer literals that can and cannot be replaced.

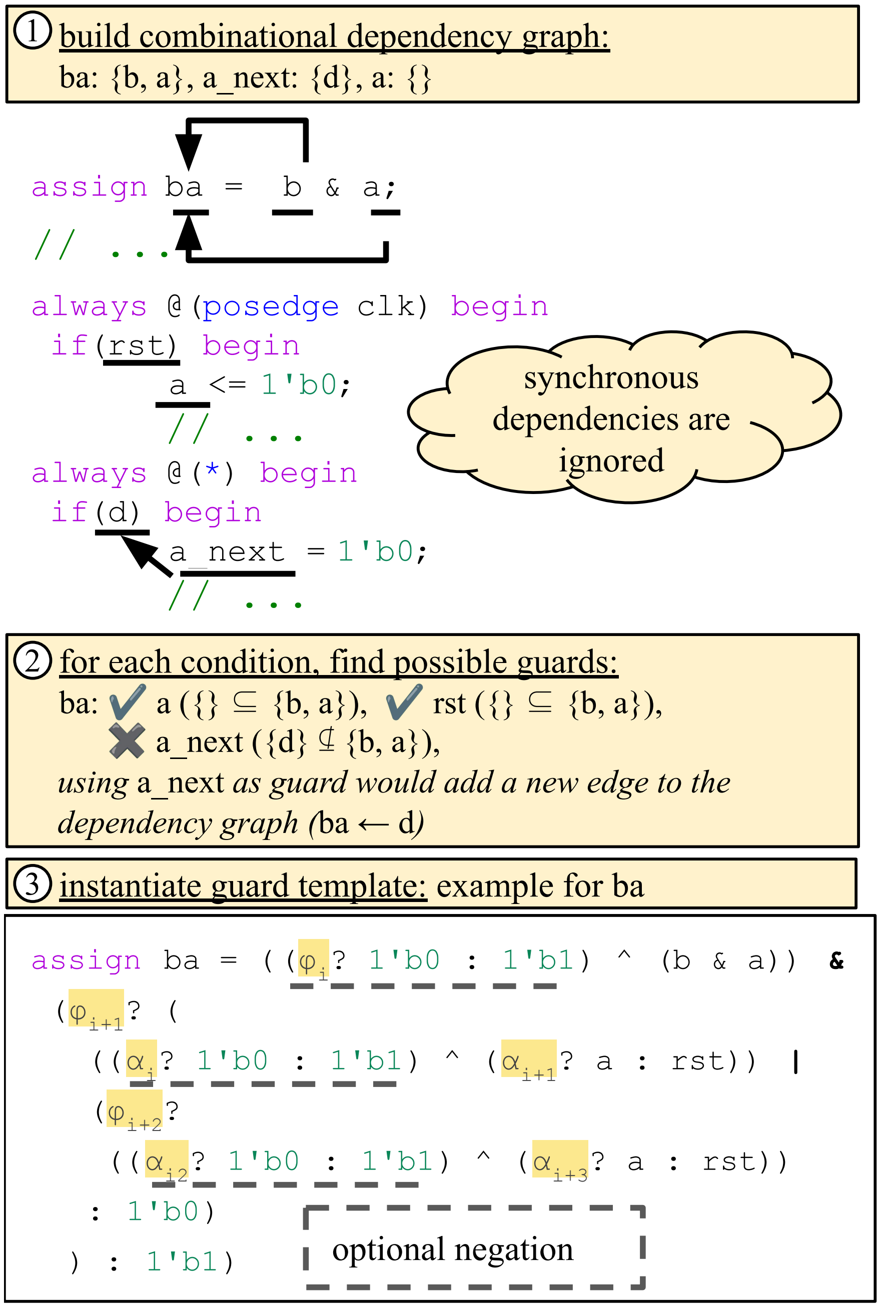


always @(posedge clk) begin
if(rst) begin
a <= 1'b0;
end else if(cnd) begin
b <= b + 1;
end
end
① analyze assignments
type: <=, vars: a, b
② extract conditions:
rst, cnd
③ create conditional assignments for each variable
if(φ_i)
if(φ_{i+1}? (α_{i+1}? rst : !rst) : 1'b1 &&
φ_{i+2}? (α_{i+2}? cnd : !cnd) : 1'b1)
a <= α_i;
for a and b
④ insert copies at start and end of process

Figure 7.4: **Conditional Overwrite Template:** Allows the repair synthesizer to assign every variable to an arbitrary constant at the start and end of every process. This assignment can be guarded by conditions mined from the same process.

**Add Guard Template.** This template allows the repair synthesizer to invert or add a guard to the condition of any if-statement or the right-hand side of any 1-bit assignment in the circuit. The transform follows this template $e \rightarrow (\neg?)e \wedge ((\neg?)a(\vee(\neg?)b)?)$, where $e$ is the original expression, $\neg?$ indicates an optional negation and $(\vee(\neg?)b)?$ an optional second part of the guard. The cost of inverting $e$ is one, the cost of adding a simple guard $\wedge a$ is one, and the cost of adding a more complex guard $\wedge(a \vee b)$ is two. For $a$ and $b$, the synthesizer is able to pick from a list of 1-bit variables that are part of the circuit. Care has to be taken not to create new combinational loops in the circuit since that would prevent us from synthesizing it. We thus first calculate all combinational dependencies in the original input circuit and then restrict $a$ and $b$ to variables that won't create any new dependencies for the left-hand side of the assignment. Figure 7.5 demonstrates our conservative approach with an example.

**Conditional Overwrite Template.** This template allows the repair synthesizer to insert new assignments of a freely chosen constant value to any signal. These assignments can happen either at the start or the end of a process and can optionally be guarded. The guard is composed of conditions extracted from the same process. Figure 7.4 shows an example. The cost of adding an unconditional assignment is one ($\alpha_i$ in Figure 7.4). Each guard within



① build combinational dependency graph:
ba: {b, a}, a_next: {d}, a: {}

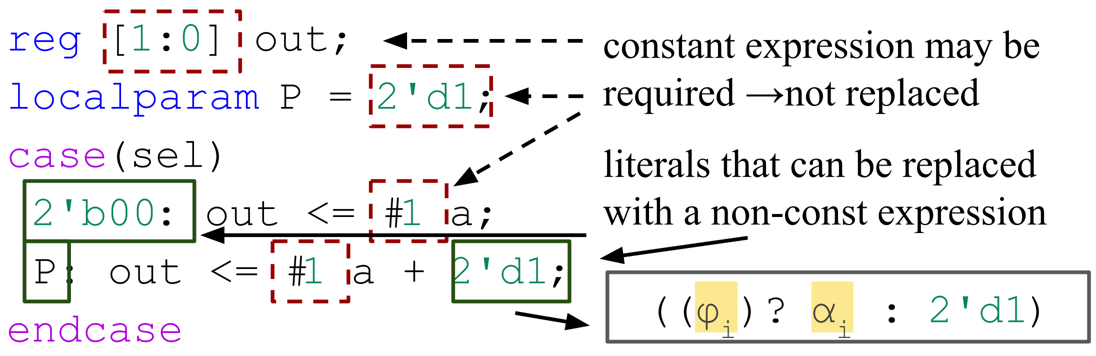
```
assign ba =  b & a;
// ...
always @(posedge clk) begin
 if(rst) begin
    a <= 1'b0;
    // ...
always @(*) begin
 if(d) begin
    a_next = 1'b0;
    // ...
```

② for each condition, find possible guards:
ba: ✓ a ({} ⊆ {b, a}), ✓ rst ({} ⊆ {b, a}),
✗ a_next ({d} ⊈ {b, a}),
*using* a_next *as guard would add a new edge to the dependency graph (*ba ← d*)*

③ instantiate guard template: example for ba

```
assign ba = ((φ_i? 1'b0 : 1'b1) ^ (b & a)) &
 (φ_i+1? (
   ((α_i? 1'b0 : 1'b1) ^ (α_i+1? a : rst)) |
   (φ_i+2?
    ((α_i2? 1'b0 : 1'b1) ^ (α_i+3? a : rst))
   : 1'b0)
  ) : 1'b1)
```

Figure 7.5: **Add Guard Template:** Allows the synthesizer to append a guard to certain 1-bit expressions. We conservatively choose possible guards to ensure that no combinational cycles are created and synthesizability is maintained.



```
reg [1:0] out;
localparam P = 2'd1;
case(sel)
 2'b00: out <= #1 a;
 P:  out <= #1 a + 2'd1;
endcase
```

```
((φi) ? αi : 2'd1)
```

constant expression may be required →not replaced
literals that can be replaced with a non-const expression

Figure 7.6: **Replace Literals Template:** Conservatively replaces literals in places where the expression is not required to evaluate to a constant at compile time.

the assignment has an additional cost of one ($\alpha_{i+1}$ and $\alpha_{i+2}$). To maintain synthesizability, our compiler pass first analyzes each process to determine which signals are assigned to in it and whether it uses blocking or non-blocking assignments. This is necessary since assigning the same signal from multiple different processes leads to race conditions in the Verilog simulation and is thus undesirable. We also want to maintain the invariant that only one type of assignment is used throughout a single process.

## Basic Synthesizer

We first discuss a very basic version of our synthesizer. The next section covers how adaptive windowing can help us scale to larger benchmarks.

**Inputs.** Our synthesizer takes in a circuit design with synthesis variables $\phi_i$ and $\alpha_i$ from the application of a repair template as well as a testbench. The design is provided in the btor2 format, which is obtained by running the synthesis tool yosys on the Verilog code. This step will fail if the design is not synthesizable. The testbench is in the format of a table with rows for each cycle of execution and columns for each input and output signal of the circuit that we are trying to repair.

**Unknown Values.** All registers start out uninitialized, and some input signals might not be defined in certain cycles of the test execution because the testbench author did not consider their value to be relevant for the test. These unknown values are modeled with Xs in Verilog. In our synthesis procedure, we either randomize or set unknown values to zero. We chose to randomize when the original testbench was using X values, as is the case for all benchmarks from CirFix. We set unknown to zero if the original testbench was using Verilator to match the behavior of that simulator.

**Synthesizing a Repair.** Having chosen concrete values for initial states and unspecified inputs, we unroll the circuit and assert that inputs and outputs have the values assigned to them from the testbench while keeping the synthesis variables $\phi_i$ and $\alpha_i$ symbolic. Then, we query the SMT-solver to obtain an assignment to the synthesis variables that will make the testbench pass. If the solver returns unsatisfiable, we know that the given template cannot repair the circuit, so we move on to the next template. If the solver returns a solution, we try to minimize the number of changes.

**Synthesizing a Minimal Repair.** We observed in our experiment that when a solution exists, the minimal solution generally only takes a small number of changes (see Table 7.5). We thus start searching for a minimal solution in a linear search, starting with one change. We encode the number of changes as a constraint into our SMT query. If a solution exists, we found a minimal solution, which is returned to the frontend to repair the Verilog code. If the solver returns unsatisfiable, we increase the number of expected changes by one. This optimization can be framed as an instance of the Max-SMT problem [15]. However, most solvers that perform well on hardware circuits do not implement Max-SMT directly. We thus stick with our customized algorithm, which allows us to use a wide range of specialized SMT solvers.

## Adaptive Windowing

As part of our basic synthesis process, we need to unroll the system once for every cycle in the testbench execution. Unfortunately, bounded model checking, and thus also our synthesis algorithm, scales poorly with how many times we unroll the system. Adaptive windowing allows us to synthesize repairs while unrolling the system for only a small number of cycles. We observed that human developers often start investigating a bug by looking at the signal values around the cycle where the first violation occurred. To make debugging tractable, developers may assume that the state of the circuit a couple of cycles before the bug manifests is correct, as this makes it simpler to trace the signal values to find a reason for the divergence. We can make use of this assumption to reduce the scope of our unrolling.

We define two values: $k_{past}$ and $k_{future}$, which specify how many cycles before and after the first output divergence we unroll our system. Our algorithm starts with both values set to zero. Thus, in the first iteration, our tool concretely executes the original circuit until the step at which the output divergence occurred and then starts the symbolic unrolling from the concrete state reached after those steps. If all state update functions are correct and there is only a bug in how the output is computed from the current state and inputs, then this would be enough to obtain a correct repair.

We generally sample all minimal repairs for a given $k_{past}$ and $k_{future}$ and then evaluate them through a concrete simulation using the repaired circuit. If the test passes, we have found a correct repair, which we return from the synthesizer. If none of the repairs work, we analyze their failures. If all of them failed at or before the same cycle as the original failure, then we assume that some state update in the past went wrong, and we need to increase the symbolic execution window towards the past. We thus increment $k_{past}$ by a constant. Generally, we chose step size two. If, on the other hand, there exists a repair that makes the earlier failure go away but then leads to a failure later in the circuit execution, we assume that we are missing some future context. Therefore, we increase $k_{future}$ so that our repair window includes the newly failing cycle. The window size is the sum of $k_{past}$ and $k_{future}$. In our RTL-Repair implementation, we set the maximum window size to 32, after which the tool will give up and declare that it cannot find a repair. We also observed that when there are many failing repairs, it generally pays off to go to a larger window size immediately. Our implementation thus advances to the next window sizer after finding four failing repairs.

We have found this new adaptive windowing technique to improve scalability for benchmarks with longer testbenches drastically. One benchmark, in particular, went from timing out after one hour to being repaired in less than ten seconds.

## 7.3 Output / State Divergence Delta

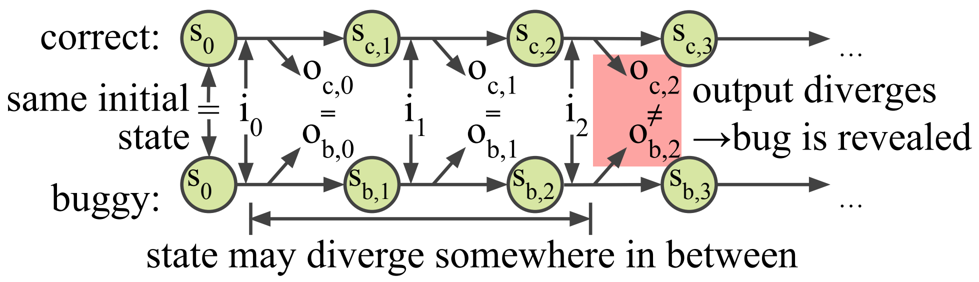
We formalize the insight behind our adaptive windowing technique through the output/state divergence delta (OSDD) metric. We assume that we are provided with a working digital synchronous circuit (the ground truth), a buggy version of the same design, a sequence of test inputs, and a starting assignment to all state variables. We then calculate the OSDD by comparing outputs and state variables on every cycle of the test execution. We note the distance between the first divergence in state values and the first divergence in output values. If the state never diverges, then the OSDD is zero. Otherwise, the OSDD is the number of steps from when the state first diverges to when the output diverges, plus one. An illustrated example of this is shown in Figure 7.7. This definition requires that the state and output variables are the same between the buggy and ground-truth versions which is true for all benchmarks that can be correctly repaired by RTL-Repair and CirFix.

We empirically calculated the OSDD by discretizing the testbench waveforms and extracting output and state (register) information from the synthesized netlist of the circuits. Our results are shown in Table 7.2. The benchmark with the largest OSDD that was successfully repaired is sdram_k2, with an OSDD of 25. However, our static analysis-based preprocessing step solved this benchmark, which does not require any unrollings (see Table 7.5). The next benchmark is i2c_k1 with an OSDD of 13, which was actually solved by the unrolling-based repair synthesizer. Both RTL-Repair and CirFix were only able to solve benchmarks with low OSDD. High OSDD benchmarks are difficult because both tools try to reason about the execution of the system.

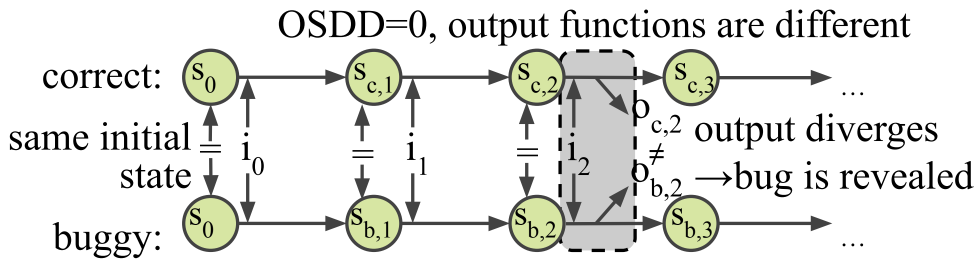
For all benchmarks repaired by RTL-Repair's synthesis engine, the OSDD provides a lower bound for how far the repair window needs to be expanded into the past. The i2c_k1 benchmark only requires a repair window of size 4, which is lower than the OSDD. While the buggy register value diverges already 12 cycles before the bug manifests, it is also updated only four cycles in the past, allowing our repair synthesizer to generate the correct repair with only 4 cycles of context. Other benchmarks require larger repair windows because future

Table 7.2: **Output / State Divergence Delta Evaluation:** Testbench (TB) length in cycles, first error (output divergence), output/state divergence delta (OSDD), size of the repair window used by RTL-Repair as well as repair results. Two i2c benchmarks are excluded as they are not clocked, and thus, the OSDD is not defined.

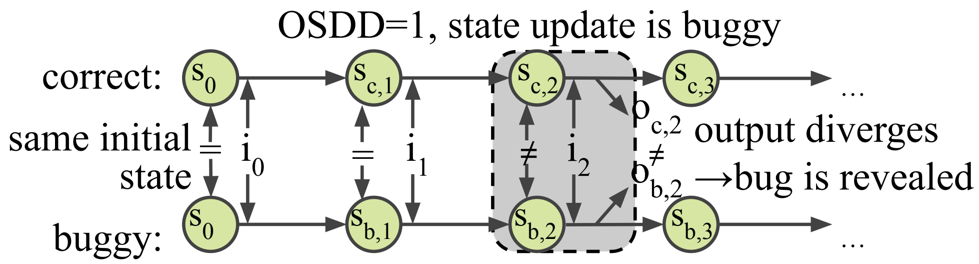
| Benchmark | TB Cycles | First Error | OSDD | Window | RTL-Repair | CirFix |
|---|---|---|---|---|---|---|
| decoder_w1 | 28 | 0 | 0 | [0 .. 10] | ✔ | ✖ |
| decoder_w2 | 28 | 0 | 0 | [0 .. 20] | ✖ | ○ |
| counter_w1 | 27 | 4 | n/a | | ○ | ✔ |
| counter_k1 | 26 | 3 | 1 | [-2 .. 0] | ✔ | ✔ |
| counter_w2 | 26 | 19 | 1 | [-2 .. 0] | ✔ | ✔ |
| flop_w1 | 11 | 0 | 1 | [-1 .. 0] | ✔ | ✔ |
| flop_w2 | 11 | 0 | 1 | [-2 .. 1] | ✔ | ✔ |
| fsm_w1 | 37 | 32 | 1 | | ○ | ○ |
| fsm_s2 | 37 | 9 | 1 | | ✔ | ✖ |
| fsm_w2 | 37 | 2 | 3 | | ✔ | ✖ |
| fsm_s1 | 37 | 10 | 11 | | ✔ | ✖ |
| shift_w1 | 27 | 8 | 1 | | ✔ | ✖ |
| shift_w2 | 27 | 0 | 1 | [-1 .. 0] | ✔ | ✔ |
| shift_k1 | 29 | 7 | n/a | | ✖ | ✔ |
| mux_k1 | 151 | 10 | 1 | | ○ | ○ |
| mux_w2 | 151 | 20 | 1 | [0 .. 10] | ✔ | ✖ |
| mux_w1 | 151 | 10 | 1 | [0 .. 20] | ✔ | ✖ |
| i2c_k1 | 171957 | 1238 | 13 | [-4 .. 0] | ✔ | ✔ |
| sha3_w1 | 357 | 24 | 1 | | ○ | ✔ |
| sha3_r1 | 357 | 24 | 1 | | ○ | ○ |
| sha3_w2 | 357 | 46 | n/a | | ○ | ○ |
| sha3_s1 | 129 | 31 | 1 | [-2 .. 0] | ✔ | ✖ |
| pairing_w1 | 74149 | 74119 | 73346 | | ○ | ○ |
| pairing_k1 | 74149 | 775 | 2 | | ○ | ○ |
| pairing_w2 | 74149 | 74119 | 74109 | | ○ | ○ |
| reed_b1 | 166166 | 2967 | 2963 | | ○ | ○ |
| reed_o1 | 166166 | 0 | 0 | | ○ | ✖ |
| sdram_w2 | 636 | 130 | 1 | [-4 .. 5] | ✔ | ○ |
| sdram_k2 | 636 | 64 | 25 | | ✔ | ○ |
| sdram_w1 | 636 | 1 | 1 | | ○ | ✖ |



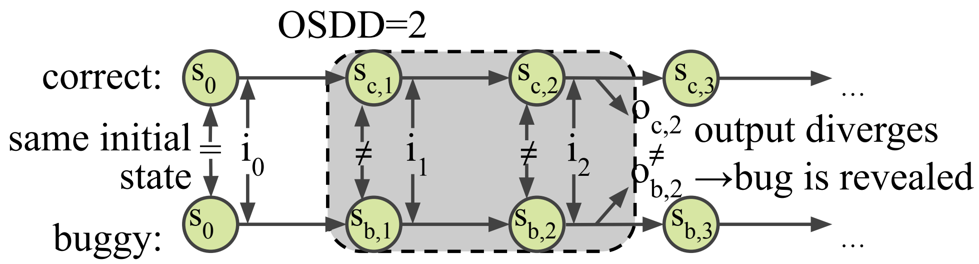
(a) We start the correct and the buggy system in the same state and execute both with the same inputs until the outputs become unequal, which happens in our example after two state updates.



(b) If all states before the output divergence are equivalent, then we define the OSDD to be zero.



(c) OSDD=1



(d) For a failure at cycle 2, the maximum OSDD is 3.

Figure 7.7: Output / State Divergence Delta (OSDD) Example

information needs to be taken into account. For example, the decoder benchmarks contain no state variables, and their OSDD is 0. However, several different inputs and, therefore, several cycles of test execution are needed in order to reveal all the bugs in the design.

## 7.4 Evaluation

We compare RTL-Repair to the prior state-of-the-art tool CirFix in terms of the quality of repairs and how quickly the repairs are provided. RTL-Repair provides more correct repairs and is often orders of magnitude faster than CirFix. We also performed a detailed analysis of the various components of RTL-Repair and how they contribute to its performance.

### Experimental Setup

All our experiments were run on a server with 252GiB of RAM and two 8-core Intel Xeon E5-2667 CPUs with hyperthreading. While the core algorithms of both RTL-Repair and CirFix could benefit from multiple cores, their current implementations are strictly sequential, and thus, multiple cores are only used to run different benchmarks in parallel to speed up our evaluation. We observed that CirFix would run slower on our machine than reported in the original paper. This could be due to the slower CPU, or VCS might have higher startup costs on our machine due to a different license server setup. We increased the timeout from 12h to 16h to ensure that CirFix has the time to generate all repairs reported by the original paper.

The RTL-Repair prototype consists of a frontend that uses the PyVerilog [137] library to implement our symbolic repair templates. The Yosys [153] tool converts Verilog designs into a transition system in the btor2 format [111]. A synthesis engine written in Rust takes the I/O trace and transition system to find a suitable repair. While the synthesis engine can work with many different SMT solvers, we use bitwuzla [109] in our experiments since it offers the best performance on average.

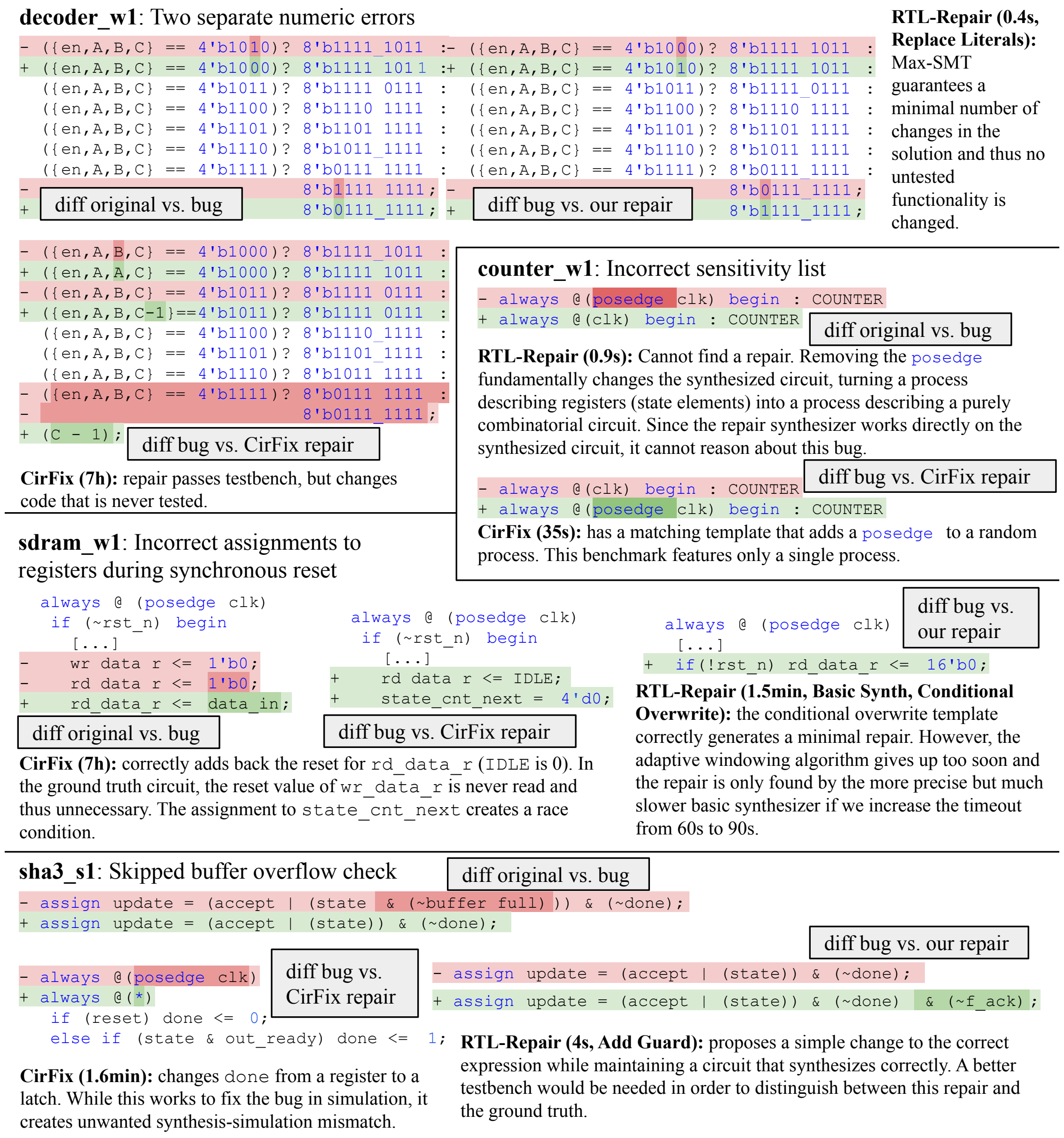
We extended the CirFix prototype [2] to allow us to run different benchmarks in parallel in order to speed up the evaluation. The core algorithm remains untouched, and our results are comparable to those reported in the CirFix paper. Table 7.3 shows the benchmarks from the CirFix paper that are used in our evaluation and maps them to the short names used throughout this paper. We created I/O traces from the provided ground truth versions of each circuit. We had to manually remove a tri-state bus and an asynchronous reset for two benchmarks as these constructs are not supported by RTL-Repair. This conversion could be automated in the future. The source code of RTL-Repair, our modified version of CirFix and all experimental scripts are available on GitHub: `https://github.com/ekiwi/rtl-repair`

Table 7.3: **Benchmark Overview**. Relates benchmarks from CirFix [2] to the short names used throughout this paper.

| Project | Defect | Short Name |
|---|---|---|
| decoder 3-8 | Two separate numeric errors | decoder_w1 |
| | Incorrect assignment | decoder_w2 |
| counter | Incorrect sensitivity list | counter_w1 |
| | Incorrect reset | counter_k1 |
| | Incorrect incremental of counter | counter_w2 |
| flip flop | Incorrect conditional | flop_w1 |
| | Branches of if-statement swapped | flop_w2 |
| fsm full | Incorrect case statement | fsm_w1 |
| | Incorrectly blocking assignments | fsm_s2 |
| | Assignment to next state and default in case statement omitted | fsm_w2 |
| | Assignment to next state omitted, incorrect sensitivity list | fsm_s1 |
| lshift reg | Incorrect blocking assignment | shift_w1 |
| | Incorrect conditional | shift_w2 |
| | Incorrect sensitivity list | shift_k1 |
| mux 4 1 | 1 bit instead of 4 bit output | mux_k1 |
| | Hex instead of binary constants | mux_w2 |
| | Three separate numeric errors | mux_w1 |
| i2c | Incorrect sensitivity list | i2c_w1 |
| | Incorrect address assignment | i2c_w2 |
| | No command acknowledgement | i2c_k1 |
| sha3 | Off-by-one error in loop | sha3_w1 |
| | Incorrect bitwise negation | sha3_r1 |
| | Incorrect assignment to wires | sha3_w2 |
| | Skipped buffer overflow check | sha3_s1 |
| tate pairing | Incorrect logic for bitshifting | pairing_w1 |
| | Incorrect operator for bitshifting | pairing_k1 |
| | Incorrect instantiation of modules | pairing_w2 |
| reed-solomon decoder | Insufficient register size | reed_b1 |
| | Incorrect sensitivity list for reset | reed_o1 |
| sdram-controller | Numeric error in definitions | sdram_w2 |
| | Incorrect case statement | sdram_k2 |
| | Incorrect assignments to registers during synchronous reset | sdram_w1 |

Table 7.4: **Repair Correctness Evaluation**
Symbols: ✔ test passed, ✖ test failed, ○ no repair to test
An empty cell means that the test did not apply. Overall a repair is judged a success (✔) if all applicable tests pass. The number in the right-most column denotes the number of changes comprising the repair.

| Benchmark | Tool | Tool Status | Testbench | CirFix Author | Gate-Level | iVerilog | Extended | Overall |
|---|---|---|---|---|---|---|---|---|
| decoder_w1 | rtlrepair | ✔ | ✔ |  | ✔ |  | ✔ | 2 ✔ |
|  | cirfix | ✔ | ✔ | ✔ | ✔ |  | ✖ | 3 ✖ |
| decoder_w2 | rtlrepair | ✔ | ✔ |  | ✔ |  | ✖ | 5 ✖ |
|  | cirfix | ○ |  |  |  |  |  | ○ |
| counter_w1 | rtlrepair | ○ |  |  |  |  |  | ○ |
|  | cirfix | ✔ | ✔ | ✔ | ✔ | ✔ |  | 1 ✔ |
| counter_k1 | rtlrepair | ✔ | ✔ |  | ✔ | ✔ |  | 1 ✔ |
|  | cirfix | ✔ | ✔ | ✔ | ✔ | ✔ |  | 5 ✔ |
| counter_w2 | rtlrepair | ✔ | ✔ |  | ✔ | ✔ |  | 2 ✔ |
|  | cirfix | ✔ | ✔ | ✔ | ✔ | ✔ |  | 1 ✔ |
| flop_w1 | rtlrepair | ✔ | ✔ |  | ✔ | ✔ |  | 0 ✔ |
|  | cirfix | ✔ | ✔ | ✔ | ✔ | ✔ |  | 1 ✔ |
| flop_w2 | rtlrepair | ✔ | ✔ |  | ✔ | ✔ |  | 0 ✔ |
|  | cirfix | ✔ | ✔ | ✔ | ✔ | ✔ |  | 2 ✔ |
| fsm_s2 | rtlrepair | ✔ | ✔ |  | ✔ | ✔ |  | 15 ✔ |
|  | cirfix | ✔ | ✔ | ✖ | ✖ | ✔ |  | 2 ✖ |
| fsm_w2 | rtlrepair | ✔ | ✔ |  | ✔ | ✔ |  | 3 ✔ |
|  | cirfix | ✔ | ✔ | ✖ | ✖ | ✔ |  | 1 ✖ |
| fsm_s1 | rtlrepair | ✔ | ✔ |  | ✔ | ✔ |  | 2 ✔ |
|  | cirfix | ✔ | ✔ | ✔ | ✖ | ✔ |  | 1 ✖ |
| shift_w1 | rtlrepair | ✔ | ✔ |  | ✔ | ✔ |  | 4 ✔ |
|  | cirfix | ✔ | ✔ | ✔ | ✔ | ✖ |  | 1 ✖ |
| shift_w2 | rtlrepair | ✔ | ✔ |  | ✔ | ✔ |  | 0 ✔ |
|  | cirfix | ✔ | ✔ | ✔ | ✔ | ✔ |  | 1 ✔ |
| shift_k1 | rtlrepair | ✔ | ✖ |  | ✖ | ✖ |  | 0 ✖ |
|  | cirfix | ✔ | ✔ | ✔ | ✔ | ✔ |  | 1 ✔ |
| mux_w2 | rtlrepair | ✔ | ✔ |  | ✔ | ✔ |  | 2 ✔ |
|  | cirfix | ✔ | ✔ | ✖ | ✖ | ✔ |  | 3 ✖ |
| mux_w1 | rtlrepair | ✔ | ✔ |  | ✔ | ✔ |  | 9 ✔ |
|  | cirfix | ✔ | ✔ | ✖ | ✖ | ✔ |  | 4 ✖ |
| i2c_w1 | rtlrepair | ○ |  |  |  |  |  | ○ |
|  | cirfix | ✔ | ✔ | ✔ |  |  |  | 1 ✔ |
| i2c_w2 | rtlrepair | ○ |  |  |  |  |  | ○ |
|  | cirfix | ✔ | ✔ | ✖ |  |  |  | 1 ✖ |
| i2c_k1 | rtlrepair | ✔ | ✔ |  |  |  |  | 1 ✔ |
|  | cirfix | ✔ | ✔ | ✔ |  |  |  | 1 ✔ |
| sha3_w1 | rtlrepair | ○ |  |  |  |  |  | ○ |
|  | cirfix | ✔ | ✔ | ✔ | ✔ | ✔ |  | 1 ✔ |
| sha3_s1 | rtlrepair | ✔ | ✔ |  | ✔ | ✔ |  | 1 ✔ |
|  | cirfix | ✔ | ✔ | ✔ | ✖ | ✔ |  | 1 ✖ |
| reed_o1 | rtlrepair | ○ |  |  |  |  |  | ○ |
|  | cirfix | ✔ | ✔ | ✔ | ✖ | ✔ |  | 2 ✖ |
| sdram_w2 | rtlrepair | ✔ | ✔ |  |  | ✔ |  | 2 ✔ |
|  | cirfix | ○ |  |  |  |  |  | ○ |
| sdram_k2 | rtlrepair | ✔ | ✔ |  |  | ✔ |  | 2 ✔ |
|  | cirfix | ○ |  |  |  |  |  | ○ |
| sdram_w1 | rtlrepair | ○ |  |  |  |  |  | ○ |
|  | cirfix | ✔ | ✔ | ✔ |  | ✖ |  | 2 ✖ |



**decoder_w1**: Two separate numeric errors

diff original vs. bug

```
- ({en,A,B,C} == 4'b1010)? 8'b1111_1011 :
+ ({en,A,B,C} == 4'b1000)? 8'b1111_1011 :
  ({en,A,B,C} == 4'b1011)? 8'b1111_0111 :
  ({en,A,B,C} == 4'b1100)? 8'b1110_1111 :
  ({en,A,B,C} == 4'b1101)? 8'b1101_1111 :
  ({en,A,B,C} == 4'b1110)? 8'b1011_1111 :
  ({en,A,B,C} == 4'b1111)? 8'b0111_1111 :
-                          8'b1111_1111;
+                          8'b0111_1111;
```

diff bug vs. our repair

```
- ({en,A,B,C} == 4'b1000)? 8'b1111_1011 :
+ ({en,A,B,C} == 4'b1010)? 8'b1111_1011 :
  ({en,A,B,C} == 4'b1011)? 8'b1111_0111 :
  ({en,A,B,C} == 4'b1100)? 8'b1110_1111 :
  ({en,A,B,C} == 4'b1101)? 8'b1101_1111 :
  ({en,A,B,C} == 4'b1110)? 8'b1011_1111 :
  ({en,A,B,C} == 4'b1111)? 8'b0111_1111 :
-                          8'b0111_1111;
+                          8'b1111_1111;
```

**RTL-Repair (0.4s, Replace Literals):** Max-SMT guarantees a minimal number of changes in the solution and thus no untested functionality is changed.

diff bug vs. CirFix repair

```
- ({en,A,B,C} == 4'b1000)? 8'b1111_1011 :
+ ({en,A,A,C} == 4'b1000)? 8'b1111_1011 :
- ({en,A,B,C} == 4'b1011)? 8'b1111_0111 :
+ ({en,A,B,C-1}==4'b1011)? 8'b1111_0111 :
  ({en,A,B,C} == 4'b1100)? 8'b1110_1111 :
  ({en,A,B,C} == 4'b1101)? 8'b1101_1111 :
  ({en,A,B,C} == 4'b1110)? 8'b1011_1111 :
- ({en,A,B,C} == 4'b1111)? 8'b0111_1111 :
-                          8'b0111_1111;
+ (C - 1);
```

**CirFix (7h):** repair passes testbench, but changes code that is never tested.

**counter_w1**: Incorrect sensitivity list

diff original vs. bug

```
- always @(posedge clk) begin : COUNTER
+ always @(clk) begin : COUNTER
```

**RTL-Repair (0.9s):** Cannot find a repair. Removing the `posedge` fundamentally changes the synthesized circuit, turning a process describing registers (state elements) into a process describing a purely combinatorial circuit. Since the repair synthesizer works directly on the synthesized circuit, it cannot reason about this bug.

diff bug vs. CirFix repair

```
- always @(clk) begin : COUNTER
+ always @(posedge clk) begin : COUNTER
```

**CirFix (35s):** has a matching template that adds a `posedge` to a random process. This benchmark features only a single process.

**sdram_w1**: Incorrect assignments to registers during synchronous reset

diff original vs. bug

```
  always @ (posedge clk)
   if (~rst_n) begin
     [...]
-    wr_data_r <= 1'b0;
-    rd_data_r <= 1'b0;
+    rd_data_r <= data_in;
```

diff bug vs. CirFix repair

```
  always @ (posedge clk)
   if (~rst_n) begin
     [...]
+    rd_data_r <= IDLE;
+    state_cnt_next = 4'd0;
```

**CirFix (7h):** correctly adds back the reset for `rd_data_r` (`IDLE` is 0). In the ground truth circuit, the reset value of `wr_data_r` is never read and thus unnecessary. The assignment to `state_cnt_next` creates a race condition.

diff bug vs. our repair

```
  always @ (posedge clk)
    [...]
+  if(!rst_n) rd_data_r <= 16'b0;
```

**RTL-Repair (1.5min, Basic Synth, Conditional Overwrite):** the conditional overwrite template correctly generates a minimal repair. However, the adaptive windowing algorithm gives up too soon and the repair is only found by the more precise but much slower basic synthesizer if we increase the timeout from 60s to 90s.

**sha3_s1**: Skipped buffer overflow check

diff original vs. bug

```
- assign update = (accept | (state & (~buffer_full))) & (~done);
+ assign update = (accept | (state)) & (~done);
```

diff bug vs. CirFix repair

```
- always @(posedge clk)
+ always @(*)
    if (reset) done <= 0;
    else if (state & out_ready) done <= 1;
```

**CirFix (1.6min):** changes `done` from a register to a latch. While this works to fix the bug in simulation, it creates unwanted synthesis-simulation mismatch.

diff bug vs. our repair

```
- assign update = (accept | (state)) & (~done);
+ assign update = (accept | (state)) & (~done) & (~f_ack);
```

**RTL-Repair (4s, Add Guard):** proposes a simple change to the correct expression while maintaining a circuit that synthesizes correctly. A better testbench would be needed in order to distinguish between this repair and the ground truth.

Figure 7.8: Qualitative comparison of RTL-Repair and CirFix repairs on four different benchmarks. The **decoder_w1** result shows how the Max-SMT-based approach can help RTL-Repair generate repairs that leave untested features untouched, while CirFix sometimes introduces new bugs. counter_w1 is a good example for a bug that RTL-Repair cannot tackle because it leads to synthesis problems. sdram_w1 shows how RTL-Repair avoids introducing new bugs by minimizing repairs.

Table 7.5: **Repair Speed Evaluation**. A direct comparison of RTL-Repair and CirFix is on the right, and the performance breakdown of the RTL-Repair components is on the left. A **bold** number indicates the number of changes performed. The Basic Synthesizer column shows the performance of RTL-Repair when benchmarks are naively unrolled without our adaptive windowing technique. Symbols: ✔ generated correct repair, ✖ generated incorrect repair, ○ no repair generated

| Benchmark | Preprocessing | Replace Literals | Add Guard | Conditional Overwrite | Basic Synthesizer | RTL-Repair | CirFix | Speedup |
|---|---|---|---|---|---|---|---|---|
| decoder_w1 | 0 0.16s | 2 ✔ 0.18s | ○ 0.09s | ○ 0.09s | ✔ 0.41s | ✔ 0.39s | ✖ 7.21h | 66,904x |
| decoder_w2 | 0 0.18s | 5 ✖ 0.26s | ○ 0.10s | ○ 0.09s | ✖ 0.59s | ✖ 0.68s | Timeout | 85,149x |
| counter_w1 | 6 0.44s | ○ 0.11s | ○ 0.13s | ○ 0.13s | ○ 0.83s | ○ 0.83s | ✔ 35.09s | 42x |
| counter_k1 | 0 0.18s | ○ 0.12s | ○ 0.08s | 1 ✔ 0.09s | ✔ 0.65s | ✔ 0.60s | ✔ 13.31h | **79,945x** |
| counter_w2 | 0 0.17s | ○ 0.20s | ○ 0.19s | 2 ✔ 0.10s | ✔ 0.67s | ✔ 0.75s | ✔ 14.19h | **67,978x** |
| flop_w1 | 0 0.18s | ○ 0.11s | 1 ✔ 0.11s | ○ 0.10s | ✔ 0.45s | ✔ 0.45s | ✔ 15.28s | **34x** |
| flop_w2 | 0 0.17s | ○ 0.11s | 2 ✔ 0.11s | ○ 0.10s | ✔ 0.44s | ✔ 0.43s | ✔ 28.57min | **3,961x** |
| fsm_w1 | 0 0.17s | ○ 0.96s | ○ 1.44s | ○ 2.65s | ○ 1.26s | ○ 5.76s | Timeout | 10,007x |
| fsm_s2 | 15 0.46s | Repaired by preprocessing | | | ✔ 0.66s | ✔ 0.65s | ✖ 2.03h | 11,191x |
| fsm_w2 | 3 0.70s | Repaired by preprocessing | | | ✔ 0.90s | ✔ 0.90s | ✖ 44.83min | 2,990x |
| fsm_s1 | 2 0.67s | Repaired by preprocessing | | | ✔ 0.90s | ✔ 0.90s | ✖ 1.11min | 73x |
| shift_w1 | 4 0.43s | Repaired by preprocessing | | | ✔ 0.58s | ✔ 0.57s | ✖ 28.58s | 50x |
| shift_w2 | 0 0.16s | ○ 0.13s | 1 ✔ 0.10s | ○ 0.11s | ✔ 0.52s | ✔ 0.46s | ✔ 35.11s | **75x** |
| shift_k1 | 0 0.17s | ○ 0.12s | | | ✖ 0.34s | ✖ 0.34s | ✔ 15.51s | 45x |
| mux_k1 | 4 0.79s | ○ 0.12s | ○ 0.08s | ○ 0.14s | ✖ 1.15s | ○ 1.28s | Timeout | 44,840x |
| mux_w2 | 0 0.17s | 2 ✔ 0.13s | ○ 0.08s | ○ 0.09s | ✔ 0.47s | ✔ 0.36s | ✖ 5.42h | 54,731x |
| mux_w1 | 6 0.58s | 3 ✔ 0.10s | ○ 0.06s | ○ 0.15s | ✔ 0.86s | ✔ 0.81s | ✖ 7.56h | 33,542x |
| i2c_w1 | 1 0.85s | ○ 0.18s | ○ 0.46s | ○ 0.31s | ○ 1.82s | ○ 1.90s | ✔ 3.86min | 122x |
| i2c_w2 | 1 0.84s | ○ 0.19s | ○ 0.41s | ○ 0.29s | ○ 1.70s | ○ 1.81s | ✖ 1.23min | 40x |
| i2c_k1 | 0 0.20s | ○ 8.52s | ○ 1.17s | 1 ✔ 3.57s | Timeout | ✔ 13.17s | ✔ 41.10min | **187x** |
| sha3_w1 | 0 0.24s | ○ 13.73s | ○ 13.89s | ○ 14.32s | ○ 31.38s | ○ 41.34s | ✔ 1.19min | 1x |
| sha3_r1 | 0 0.22s | Timeout | ○ 0.36s | ○ 0.34s | Timeout | Timeout | Timeout | 964x |
| sha3_w2 | 0 0.24s | ○ 0.36s | ○ 0.64s | ○ 22.49s | ○ 34.80s | ○ 21.78s | Timeout | 2,644x |
| sha3_s1 | 0 0.20s | ○ 3.27s | 1 ✔ 0.33s | ○ 10.15s | ✔ 5.98s | ✔ 3.77s | ✖ 1.60min | 25x |
| pairing_w1 | 0 18.49s | Timeout | ○ 41.20s | ○ 45.44s | Timeout | Timeout | Timeout | 963x |
| pairing_k1 | 0 18.46s | Timeout | ○ 41.79s | ○ 42.03s | Timeout | Timeout | Timeout | 963x |
| pairing_w2 | 0 18.42s | Timeout | ○ 41.11s | ○ 41.24s | Timeout | Timeout | Timeout | 963x |
| reed_b1 | 0 0.33s | ○ 1.79s | ○ 1.09s | ○ 1.81s | ○ 5.44s | ○ 5.52s | Timeout | 10,428x |
| reed_o1 | 0 0.36s | ○ 1.35s | ○ 0.85s | ○ 0.89s | ○ 3.59s | ○ 3.63s | ✖ 9.50h | 9,426x |
| sdram_w2 | 0 0.18s | 2 ✔ 2.27s | ○ 0.37s | ○ 25.56s | Timeout | ✔ 2.59s | Timeout | 22,231x |
| sdram_k2 | 2 0.83s | Repaired by preprocessing | | | ✔ 1.17s | ✔ 1.20s | Timeout | 48,157x |
| sdram_w1 | 0 0.18s | ○ 0.32s | ○ 0.38s | ○ 0.62s | Timeout | ○ 1.65s | ✖ 6.91h | 15,055x |

## Quality of Repairs

The most important metric for a repair tool is the number of bugs it can successfully repair. This requires us to classify any repair the tool comes up with as correct or incorrect. The authors of CirFix followed a two-step approach: (1) By design, all repairs that CirFix returns pass the provided testbench. These repairs were described to be "plausible". (2) In a second step, the first author of the paper would manually inspect each "plausible" repair and determine whether the repair is "correct". [1] We also inspected the repairs CirFix performed and found that many seemed incorrect to us.

Many of these disagreements relate to what each research team focuses on repairing. It appears that the CirFix authors are focused on repairing the Verilog simulation of a circuit, which CirFix accomplishes in many cases. However, the goal of RTL-Repair is to repair the circuit that is described by the Verilog simulation and not just the simulation itself. Under this framing, repairs that fix the simulation but lead to synthesis-simulation mismatch (see Section 2.2) are incorrect. Since these mismatches are notoriously difficult to debug, CirFix might cause more work than it saves.

A common way to detect synthesis-simulation mismatch is so-called gate-level simulation. For this purpose, we take the output of our synthesis tool in the form of a low-level Verilog description and plug it into the original testbench. Sometimes gate-level simulation fails, not because of an actual mismatch but because of various X-propagation issues. Therefore we only perform the gate-level simulation check if it works with the ground truth version of the circuit. We add another automated check for simulator compatibility: If the original circuit works with the open-source iverilog simulator [151], the repaired version should also work with iverilog. This helps us filter out repairs that rely on race conditions or otherwise ill-defined Verilog features.

The importance of avoiding synthesis-simulation mismatch is illustrated by the mux_w1 benchmark, which CirFix repairs through a "clever" combination of blocking and non-blocking assignments. A value is overwritten by a non-blocking statement, which appears in the program order *before* the blocking statement, which assigns the original default value. This repair fixes the simulation but is not correctly understood by the synthesis tool, leading to a much harder-to-detect and debug problem for the developer to deal with.

Finally, we noticed a problem with the testbench accompanying the decoder benchmarks. It does not adequately test all functionality of the design. We thus added an extended testbench that tests all relevant input combinations. This test shows one of the advantages of minimizing the number of repairs in the RTL-Repair algorithm: It ensures that RTL-Repair only changes code exercised by the testbench. CirFix, on the other hand, ends up destroying functional parts of the circuit that were not exercised by the testbench. The second decoder benchmark contains errors in parts of the design that were never tested by the original testbench and thus cannot be repaired by any tool. If we provide RTL-Repair with the extended testbench, it successfully finds the complete repair.

[1]Source: personal communication with the authors.

Table 7.6: RTL-Repair results for bugs from open-source projects collected by the authors of "Debugging in the Brave New World of Reconfigurable Hardware" (Table 2 in [90]). All results were obtained using the incremental synthesizer and with a timeout of 2min. "Bug Diff" indicates how many lines need to be added or removed in order to go from the repaired to the buggy version of the circuit. "TB" shows the number of steps in the provided testbench.

| | Bug Diff | TB | Result and Quality | | | Template |
|---|---|---|---|---|---|---|
| D4 | +27 / -26 | 185 | Timeout | | | |
| D8 | +2 / -2 | 14 | **1** ✔ | 1.06s | B | Replace Literals |
| D9 | +2 / -2 | 523k | Timeout | | | |
| D11 | +0 / -2 | 17 | **1** ✔ | 54.12s | C | Cond. Overwrite |
| D12 | +1 / -1 | 16 | **1** ✔ | 6.06s | D | Replace Literals |
| D13 | +1 / -3 | 6 | **3** ✔ | 1.54s | C | Cond. Overwrite |
| C1 | +1 / -1 | 523k | **1** ✔ | 33.17s | A | Add Guard |
| C3 | +1 / -7 | 523k | ○ | 21.56s | | |
| C4 | +1 / -1 | 10 | **1** ✔ | 1.83s | A | Add Guard |
| S1.R | +1 / -1 | 10 | **1** ✔ | 10.23s | C | Add Guard |
| S1.B | +2 / -2 | 10 | **2** ✔ | 9.09s | D | Add Guard |
| S2 | +1 / -2 | 45 | **1** ✔ | 0.73s | C | Replace Literals |
| S3 | +12 / -35 | 13 | **2** ✔ | 6.89s | D | Replace Literals |

Overall, RTL-Repair finds 16 repairs that pass all our tests, while CirFix finds 10. Figure 7.8 features a qualitative comparison of four benchmarks, highlighting the strengths and weaknesses of both tools. RTL-Repair also provides only two incorrect repairs. The first one is due to the shortcomings in the decoder testbench. For the shift_k1 benchmark, RTL-Repair incorrectly determines that no repair is necessary since the synthesized circuit looks correct. This could easily be filtered out by running the original testbench once after a successful repair. With our more extensive testing in place, we notice that only two multi-edit repairs generated by CirFix are considered correct (counter_k1 and flop_w2). This calls into question CirFix's ability to generate multi-edit repairs that take full advantage of the genetic algorithm.

## Repair Speed

Table 7.5 shows how long RTL-Repair and CirFix take for each repair. We used a timeout of 60 seconds for RTL-Repair and 16 hours for CirFix. RTL-Repair generally provides results in a small number of seconds, often several orders of magnitude faster than CirFix. It gives almost instant feedback allowing a user to quickly decide whether they want to use the repair suggestion.

We compare the adaptive windowing technique used by RTL-Repair to the basic synthesizer. For benchmarks with small testbench lengths, the basic synthesizer is faster. But for longer testbenches like the mux benchmarks, the adaptive windowing approach leads to faster results. It allows us to solve two more benchmarks compared to the basic synthesizer.

Under normal circumstances, RTL-Repair tries out repair templates in sequence and immediately returns as soon as a repair is found. The left half of table 7.5 shows what happens if we turn off this early exit. We can see that the repair templates do not overlap; only a single repair template per benchmark generates a repair. Each repair template fixes between three and four of the benchmarks, demonstrating that the templates are not specific to a single bug. We can also see that the number of changes for each repair is small. Most often, only one or two changes are enough; the maximum is three changes to generate a correct repair. Five benchmarks are fixed directly by our static-analysis-based preprocessing phase, demonstrating the importance of combining static analysis with more sophisticated repair techniques.

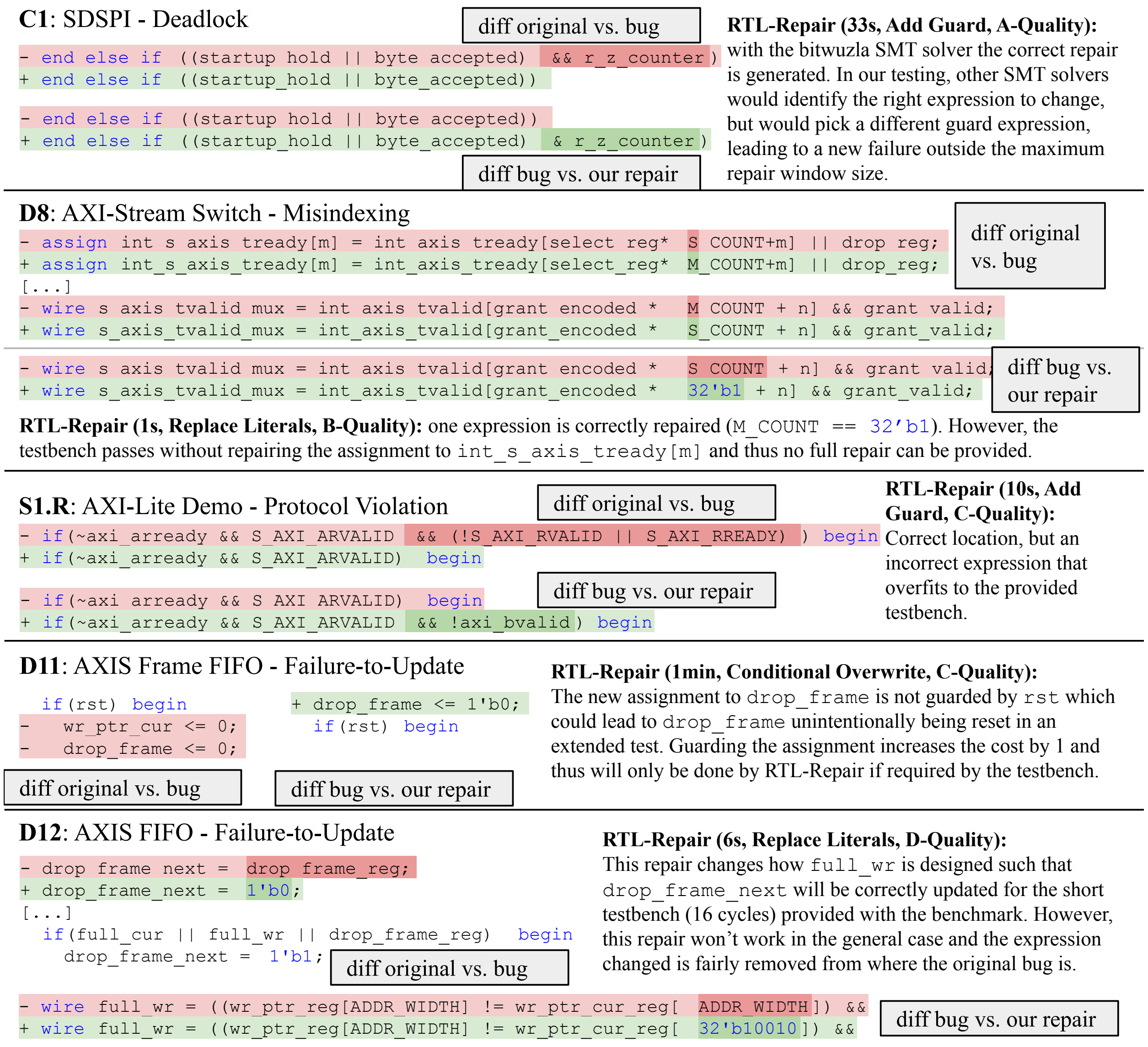
## Open-Source Bug Repair

In addition to the CirFix benchmarks, which were specifically created to test automated repair tools, we also applied our RTL-Repair tool to a set of bugs mined from git commits to open-source FPGA hardware projects [90]. Of the 20 reproducible bugs provided by the prior work [90], we are able to use 12 with RTL-Repair. The other 8 contain non-synthesizable Verilog, use SystemVerilog features that our parser is not able to deal with, or lack a ground-truth repair.

Table 7.6 shows our results. Overall, RTL-Repair provides repairs that pass the provided testbench for 9 out of 12 bugs. However, since this set of bugs was never intended to be used as a benchmark for automated repair, most testbenches are quite minimalistic and only enough to demonstrate the bug. We thus manually inspect each repair and rate it on the following scale: (A) repair matches the ground truth exactly, (B) repair performs some of the changes from the ground truth, (C) repair changes the same expression as the ground truth but in a different way, and (D) change is very different from the ground truth. Figure 7.9 shows several example repairs.

To tackle his new challenging benchmark set, we needed to improve our repair templates to make them more powerful. The Add Guard template, for example, was previously used to allow only the inversion of boolean conditions. We added the ability to add another boolean condition as a guard. While we had to improve our templates, we were still able to implement them in under 150 lines of Python each and keep the number of templates at three.

While we did improve our templates, the core synthesis algorithm remained largely untouched. This shows that while templates need to be carefully engineered to work across a large set of repair scenarios, the basic synthesis technique proposed in this paper can be applied to a wide range of designs. Note that none of these more realistic benchmarks struggled with synthesis-simulation mismatch, and none were repaired by preprocessing alone. However, while the bugs are all mined from open-source projects, most only come with ar-


diff original vs. bug
diff bug vs. our repair

**C1**: SDSPI - Deadlock

```
- end else if ((startup hold || byte accepted) && r z counter)
+ end else if ((startup_hold || byte_accepted))

- end else if ((startup hold || byte accepted))
+ end else if ((startup_hold || byte_accepted) & r_z_counter)
```

**RTL-Repair (33s, Add Guard, A-Quality):** with the bitwuzla SMT solver the correct repair is generated. In our testing, other SMT solvers would identify the right expression to change, but would pick a different guard expression, leading to a new failure outside the maximum repair window size.

**D8**: AXI-Stream Switch - Misindexing

```
- assign int s axis tready[m] = int axis tready[select reg* S COUNT+m] || drop reg;
+ assign int_s_axis_tready[m] = int_axis_tready[select_reg* M_COUNT+m] || drop_reg;
[...]
- wire s axis tvalid mux = int axis tvalid[grant encoded * M COUNT + n] && grant valid;
+ wire s_axis_tvalid_mux = int_axis_tvalid[grant_encoded * S_COUNT + n] && grant_valid;

- wire s axis tvalid mux = int axis tvalid[grant encoded * S COUNT + n] && grant valid;
+ wire s_axis_tvalid_mux = int_axis_tvalid[grant_encoded * 32'b1 + n] && grant_valid;
```

**RTL-Repair (1s, Replace Literals, B-Quality):** one expression is correctly repaired (`M_COUNT == 32'b1`). However, the testbench passes without repairing the assignment to `int_s_axis_tready[m]` and thus no full repair can be provided.

**S1.R**: AXI-Lite Demo - Protocol Violation

```
- if(~axi_arready && S_AXI_ARVALID && (!S_AXI_RVALID || S_AXI_RREADY) ) begin
+ if(~axi_arready && S_AXI_ARVALID) begin

- if(~axi arready && S AXI ARVALID) begin
+ if(~axi_arready && S_AXI_ARVALID && !axi_bvalid) begin
```

**RTL-Repair (10s, Add Guard, C-Quality):** Correct location, but an incorrect expression that overfits to the provided testbench.

**D11**: AXIS Frame FIFO - Failure-to-Update

```
  if(rst) begin
-   wr_ptr_cur <= 0;
-   drop_frame <= 0;
```

```
+ drop_frame <= 1'b0;
  if(rst) begin
```

**RTL-Repair (1min, Conditional Overwrite, C-Quality):** The new assignment to `drop_frame` is not guarded by `rst` which could lead to `drop_frame` unintentionally being reset in an extended test. Guarding the assignment increases the cost by 1 and thus will only be done by RTL-Repair if required by the testbench.

**D12**: AXIS FIFO - Failure-to-Update

```
- drop frame next = drop frame reg;
+ drop_frame_next = 1'b0;
[...]
  if(full_cur || full_wr || drop_frame_reg) begin
    drop_frame_next = 1'b1;

- wire full wr = ((wr ptr reg[ADDR WIDTH] != wr ptr cur reg[ ADDR WIDTH]) &&
+ wire full_wr = ((wr_ptr_reg[ADDR_WIDTH] != wr_ptr_cur_reg[ 32'b10010]) &&
```

**RTL-Repair (6s, Replace Literals, D-Quality):** This repair changes how `full_wr` is designed such that `drop_frame_next` will be correctly updated for the short testbench (16 cycles) provided with the benchmark. However, this repair won't work in the general case and the expression changed is fairly removed from where the original bug is.

Figure 7.9: Repairs produced by RTL-Repair for the Open-Source bugs discussed in Section 7.4.

tificially short testbenches that are provided only to demonstrate each bug. This leads to many possible repairs that can make the testbench pass. While RTL-Repair always provides a very small repair, some of them are not very good. Sampling multiple repairs and presenting them to the user could be a future fix to this problem.

## 7.5 Discussion

RTL-Repair clearly illustrates the power of symbolic analysis-based repair techniques, providing more correct repairs – orders of magnitude faster than the generate-and-validate based CirFix tool. We carefully designed RTL-Repair to work with the exact same assumptions as the prior work to make it a drop-in replacement for CirFix. This shows that symbolic repair does not require formal specifications.

Our repair templates are directly applied to the Verilog AST, making it trivial to map the repair suggested by the synthesizer back to the original design. Initially, we explored templates that worked on the transition system representation, which led to repairs that proved difficult to automatically incorporate into the high-level Verilog code. Because of our standardized interface to the repair synthesizer, new templates are easy to add.

We introduce gate-level simulation as a new standard for evaluating automated repairs of hardware designs. This ensures that the users of these tools are not in for a bad surprise when the automated repair makes their Verilog simulation work but then leads to silent bugs in the actual circuit when it is mapped to an FPGA or taped out in a VLSI process.

RTL-Repair provides repair suggestions in a matter of seconds. Through our adaptive windowing technique, this remains true, even for larger benchmarks. With this level of responsiveness, we imagine that RTL-Repair could be integrated into a Verilog IDE to directly provide quick repair suggestions, similar to tools like GitHub Copilot [53]. This would require more research into how exactly RTL-Repair could be integrated with various forms of testbenches and formal tests.

# Chapter 8

# Related Work

This chapter discusses related work for coverage-directed test generation and automated hardware design repair.

## 8.1 Coverage-Directed Fuzz Testing of RTL

My work on coverage-directed fuzz testing of RTL and the RFUZZ tool was informed by much prior art on generating new test inputs for circuit designs from coverage feedback. Since my work was published in 2018, it has inspired numerous followup papers by research groups worldwide. This section discusses both the prior art and followup work.

### Prior Art on Coverage-Directed Test Generation for Hardware

The prior work most similar to RFUZZ is MicroGP [133], which focuses on maximizing statement coverage in the HDL description of various processor implementations. It uses an instruction template library to generate and recombine programs, which allows for more powerful mutation techniques but increases the amount of setup work needed. It also restricts this line of work to the domain of processor testing, whereas coverage-directed fuzz testing also shows promising results when testing various communication IPs. Another difference is that MicroGP targets slow DUT execution in a software simulation. An industrial evaluation reports that simulation takes 30 times more resources than the core genetic algorithm [65]. RFUZZ, on the other hand, is geared towards fast FPGA-accelerated simulation, using a simpler algorithm to keep up with the test execution speed provided by such a platform.

Various other approaches to the CDG problem do not rely on a modified genetic algorithm. Tarsiran et al.[138] analyze the circuit to improve the biases for an existing random input generator. Our work, on the other hand, does not assume that a generator exists. Nativ et al. [106] propose a system that directs a random input generator with coverage feedback. However, this system also relies on rules unique to a single DUT that need to be specified by a verification expert. Fine et al. [49] present coverage-directed test generation using Bayesian Networks to guide the input generation. While the system automatically learns the network weights, the network topology is DUT-specific and needs to be designed by a verification engineer. Wang et al. [148] use a manually designed abstract model of the DUT to generate inputs that maximize coverage automatically.

`bluecheck` [107] is a synthesizable test bench framework that takes advantage of the BlueSpec HDL. Similar to our work, it is designed for FPGA-based testing. However, the authors report some issues reproducing failing test cases discovered with the FPGA emulation. Our system employs the *MetaReset* and *SparseMem* techniques to ensure that the FPGA-accelerated simulation results are deterministic. While `bluecheck` depends on the BlueSpec HDL, RFUZZ is HDL-agnostic. Our work focuses only on maximizing RTL coverage whereas `bluecheck` can also generate checks to find design bugs while running on the FPGA.

## CPU Fuzzing

After my work on RFUZZ, I was disappointed that we could not find any bugs with it due to a lack of a *checker* component that would detect if a fuzzer-generated input triggers an invalid execution. In the software domain, numerous dynamic instrumentation techniques can catch common bugs as they happen. A popular technique is to use address sanitizer [128], which will crash the program if a memory safety violation is detected. While no such common dynamic analysis techniques are available for hardware, the authors of "DIFUZZRTL: Differential Fuzz Testing to Find CPU Bugs" [60] realized that by focusing their efforts on fuzz testing CPUs instead of arbitrary RTL designs, they could simply compare the outcome of executing a generated assembly program on the RTL implementation and executing the same program on an existing functional reference simulator. This differential testing technique is common in CPU verification, but I was unaware before the DiFuzz paper was published.

Besides developing a working *checker* component for hardware fuzz testing, the DiFuzz authors also wholly redesigned the *input generator*. Instead of the byte-level, AFL-style approach used by RFUZZ, they designed a random RISC-V program generator and a separate RISC-V program mutator. They also defined a new coverage metric. Instead of checking how many mux control signals toggle, they check whether control registers take on a never-seen-before value. This new technique expands the number of generated inputs considered interesting and thus will be saved for the fuzzer to mutate. However, as reported in a later paper [25], this detailed metric leads to the fuzzer saving the vast majority of inputs it generates. Thus the new coverage metric may actually be too sensitive, overwhelming the fuzzer with too many inputs to be mutated. The authors also report that their new coverage metric is much more efficient to implement than the one used by RFUZZ. However, whether that is just an artifact of how I implemented the coverage collection for RFUZZ is unclear. The overhead of collecting the mux toggle coverage on an FPGA could be significantly reduced using the scan-chain implementation that I described in Chapter 3 of this thesis. The DiFuzz authors' idea to use a differential test oracle was brilliant, leading to many actual bugs found.

However, it is unclear how much of the improvement over RFUZZ is due to the new coverage metric and how much is due to the hand-crafted, RISC-V-specific input generator.

The following paper in this line of work [72] presents a tool called *TheHuzz* and argues that we should employ even more coverage metrics beyond the mux toggle coverage used in RFUZZ and the register coverage in DiFuzz. *TheHuzz* uses branch, condition, FSM, expression, and toggle coverage as feedback. There are generally two downsides that need to be considered when adding new coverage feedback: (1) the slowdown in fuzzing speed when adding more detailed coverage and (2) whether the increase in inputs selected for mutation means that there is not enough time for the fuzzer to spend to mutate each input. Unfortunately, the evaluation uses the number of instructions executed in simulation instead of time and does not account for simulation speed. The authors report that they faced increased overhead for the coverage metrics they use. However, they never explore whether turning off some metrics would increase overall fuzzer performance. *TheHuzz* also uses an entirely new RISC-V instruction generator and mutator, making a detailed comparison to DiFuzz difficult.

A followup paper from the same group explores using a model checker to improve the performance of the *TheHuzz* fuzzer [28]. Similar to what I report in Chapter 6, they find that simply using a formal tool to generate inputs can beat a fuzzer. Their tool makes use of a model-checker to find inputs that reach uncovered points and then provides these inputs as seeds to a version of the *TheHuzz* fuzzer. This combined approach leads to better performance than the individual techniques on their own. While prior papers use various coverage metrics to compare fuzzer performance, this paper compares the time it takes various tools to re-discover known bugs in the tested designs. This metric is interesting since it is pretty independent of the automated input generation tool's approach and avoids questions about which coverage metric should be used.

While the authors of DiFuzz and *TheHuzz* focus on expanding the coverage feedback metrics to include more details, the paper "ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance" [25] argues that using RTL design coverage as a feedback metric is misguided. Their evaluation shows that DiFuzz without any feedback achieves better coverage than DiFizz with coverage feedback (Figure 6a in the paper). This demonstrates that the feedback metrics were not as important as claimed by previous papers. The overall better performance might be due to the RISC-V instruction generator of *TheHuzz* outperforming the one used by DiFuzz. The paper also reports that DiFuzz considers more than half of the inputs it generates as interesting (Figure 6b), a very high amount. In software fuzzing, only a tiny number of inputs are considered interesting, and elaborate scheduling methods have been proposed to spend more time mutating promising inputs [17, 84] and essentially discarding less promising ones. Instead of relying on RTL coverage, the authors of ProcessorFuzz propose a custom coverage metric based on values in the CSRs when running the generated input in a functional simulator. If the input does not improve coverage, it can thus immediately be discarded without running a costly RTL simulation, leading to improved performance over DiFuzz. Unfortunately, the source code of *TheHuzz* is not available, and thus, no direct comparison could be made.

We have seen a shift away from using RTL coverage in CPU fuzzing. "Cascade: CPU

Fuzzing via Intricate Program Generation" [132] published this year argues that no coverage feedback is needed and we should instead focus on better input generation. Their advanced program generator uses a functional simulator to execute instructions as they are generated and, this way, obtain much better test programs. They show a significant speedup of their approach compared to numbers from the *TheHuzz* paper. I have gleaned from personal conversations that big processor companies also maintain sophisticated program generators to test their CPUs without a feedback-directed component. Their generators also incorporate a functional simulator to improve the quality of the programs they generate. Thus, this work seems to have brought us full circle, from feedback-directed mutational fuzz testing, which had only been tried in academia, to re-discovering the constraint-random program generators that have been standard in the industry for years. This is not a criticism of the authors since they provide valuable data on the effectiveness of this approach.

The authors of "GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs" [86] explore using GPUs to execute many fuzzer-generated inputs in parallel. They also implement a new mutation algorithm, much closer to a traditional genetic algorithm that ranks generated inputs each generation and discards all besides the top ones compared to an AFL-style algorithm, which never discards inputs deemed interesting. Unfortunately, the evaluation mainly provides end-to-end results, making it hard to determine how much of the improved speed is due to the new mutation algorithm and how much is due to using GPUs for execution.

While all other papers in this section use the fuzzer to generate RISC-V programs to test their CPUs, the authors of "Effective Processor Verification with Logic Fuzzer Enhanced Co-simulation" [71] instead use the fuzzer to change how the CPU executes an existing test suite of RISC-V programs. The authors observe that certain parts of the CPU design should be resilient to random changes. Handshake-based interfaces should allow for backpressure to be asserted at arbitrary times. By allowing the fuzzer to control these interfaces within their specification, the tool discovers bugs through stress testing the design.

## Fuzzing with AFL

While RFUZZ re-implemented large parts of the AFL algorithm, various papers explored using AFL directly to fuzz RTL designs. While this decreases the flexibility in designing hardware-specific input generators and feedback metrics, it can be a powerful approach to set up a basic fuzzer quickly.

The authors of "Hyperfuzzing for SoC security validation" [104] are the first to use Verilator to convert the RTL of the design under test into a C++ simulation and then apply AFL to fuzz the resulting program. They are also the first to explore using more sophisticated input generators that translate the byte stream generated by AFL into high-level actions that exercise the hardware. This work calls these components *adverserial state tamperers*. The idea of using generators in this way was first introduced for software fuzzing by the Zest paper [115]. The authors of Hyperfuzzing use a custom high-level coverage feedback metric, which is communicated directly to AFL. The evaluation and comparison with RFUZZ suffers from a misunderstanding: Mux toggle coverage only measures whether a mux control signal toggles; the toggle coverage built into Verilator, on the other hand, measures whether any signal in the design toggles, leading to much higher overhead. The considerable slowdown observed by the authors when enabling full toggle coverage tracks our results reported in Chapter 3.

"Fuzzing hardware like software" [141] popularized the use of AFL for hardware fuzzing. Like the Hyperfuzzing paper, the authors use Verilator to generate C++ source code for fuzzing. However, instead of devising their custom coverage metric, they show that AFL's default instrumentation, which tracks coverage of the edges in the program's control flow graph, correlates well with RTL line coverage. Thus, they can use AFL with no modifications. Similar to the previous work, the authors also employ generators but view them through the lens of grammar-based fussing. Their generators target TileLink bus components and translate AFL-generated byte streams into valid TileLink transactions.

The authors of "Efficient Cross-Level Processor Verification using Coverage-guided Fuzzing" also use AFL to fuzz C++ code generated by Veriator from an RTL design [22]. They are the first to link a SytemC-based functional simulator into the generated binary, allowing AFL to use coverage from both the RTL and the functional model as feedback. They also augment AFL by adding known RISC-V instructions as seed inputs to improve its effectiveness. The paper does not compare to other CPU fuzzing tools. "SpinalFuzz: Coverage-Guided Fuzzing for SpinalHDL Designs" [123] uses AFL in a fashion that is very similar to "Fuzzing hardware like software". They integrate fuzz harness generation and fuzzer execution into the SpinalHDL testing framework to offer a seamless user experience. They report results similar to RFUZZ with the feedback-directed fuzz testing generating better coverage than a random baseline.

## Other Hardware Fuzzing Approaches

The authors of "DirectFuzz: Automated Test Generation for RTL Designs using Directed Graybox Fuzzing" [24] adapt RFUZZ to target specific submodules. It is the only paper that re-uses the RFUZZ benchmarks. The overall coverage achieved by this approach is not better than what RFUZZ achieves, but the new fuzzer obtains this slightly faster for the targeted module. Considering how the benchmarks used by RFUZZ are not very good, as discussed in Chapter 6, this small improvement is probably irrelevant.

The authors of "Symbolic Simulation Enhanced Coverage-Directed Fuzz Testing of RTL Design" [85] combine symbolic execution and fuzzing, interleaving them in a fashion similar to the software fuzzing tool Driller [134]. The benchmark set differs from any other fuzzing tool, so it is hard to compare. An interesting further research question is whether using bounded model checking to generate seeds as proposed in "HyPFuzz: Formal-Assisted Processor Fuzzing" [28] would be more effective than the symbolic execution approach employed in this work.

While it does not use feedback-directed fuzzing, the work on "PyH2: Using PyMTL3 to Create Productive and Open-Source Hardware Testing Methodologies" [68] is noteworthy as it is the only one to allow for easy test case reduction. As we observed in Chapter 6, randomly generated inputs are often unnecessarily complex. Using the established Python property testing framework Hypothesis [91], PyH2 can use its in-built test case reduction techniques.

## 8.2 Automated Repair of Hardware Design Code

The RTL-Repair tool, which I discussed in Chapter 7 and the CirFix that inspired my work are currently the only end-to-end tools that generate complete repairs from a buggy Verilog source code and testbench alone. Recent work using LLMs assumes the precise fault location is known [1]. There are many hardware fault localization approaches, but none are exact [70, 120, 154]. Some other repair tools rely on C reference models [4] or formal LTL properties [36, 16] instead of testbenches.

There has been work on symbolic-analysis-based repair for hardware [16, 92, 27]. However, none of these approaches can deal with long-running testbenches; instead, they focus on bugs that appear after one or two execution cycles. The work by *Chang et.al.* [27] is noteworthy because it uses a two-step approach that first identifies faulty expressions and then synthesizes a repair to replace them. A similar approach was independently discovered years later for software repair with the Angelix tool [100, 108, 26].

# Chapter 9

# Conclusion

This thesis presents four tools, each improving the modern RTL development experience. Much recent effort has been spent on new hardware languages promising to improve developer productivity. This thesis instead focuses on how we can design better testing and debugging tools. Each of the four tools discussed exemplifies some of the challenges and opportunities in this new age of open-source hardware languages and RTL design tools.

New hardware languages promise to improve RTL designer productivity. However, the associated tools often lack important debugging features commonly available in more mature language ecosystems. Since these new languages come with open-source compilers, we can apply ideas from software compilers to implement debugging features more efficiently than ever before. For example, Chapter 3 introduces a new approach for implementing coverage feedback that decouples the hardware language-specific coverage instrumentation from the simulator used to execute the low-level hardware description. A simple new `cover` statement is easy to support across a range of very different hardware simulators – from interpreter to FPGA-accelerated simulation – and makes it possible to implement new coverage metrics that work across all simulators by design.

Implementing automated coverage in an open-source compiler has benefits beyond reduced implementation complexity and broad simulator support. Chapter 4 examines how the coverage information can be used to generate novel and interesting inputs to a design under test. To this end, we adopt feedback-directed mutational fuzz testing from the software domain to work with arbitrary RTL designs. Besides defining a new mux-toggle coverage metric, RFUZZ proposes a novel mapping of fuzzer-generated input bytes to circuit inputs and a new approach to dealing with input constraints, which are much more common in hardware than in software. To speed up fuzz testing, RFUZZ incorporates new isolation techniques to enable FPGA acceleration. RFUZZ inspired numerous works on improved hardware fuzz testing, the most important of which are summarized in Chapter 8.

In the previous chapter, we used coverage feedback from simulation to generate inputs that reveal bugs in hardware designs through fuzz testing. Formal methods like bounded model checking (BMC) instead model the design under test in first-order logic and then ask an SMT solver if an input exists that would violate an assertion. Chapter 5 details how

I implemented BMC for the novel Chisel hardware construction languages. It shows how the hardware generator approach and the focus on unit testing with Chise can seamlessly be applied to formal techniques. Once again, this chapter highlights how an open-source compiler can be leveraged to quickly implement a new formal backend, which I carefully designed to enable accurate simulation replay, allowing users to use printf-style debugging for the formal tests.

Both fuzz testing and BMC show promising results. However, they are often only considered in isolation. Chapter 6 investigates the question of when each technique is the most appropriate to apply. It shows that the answer is often hard to predict. Instead, I advocate for always applying at least a simple baseline version of BMC and random testing when proposing a more sophisticated technique. I also present how automated input generation setups can be divided into three components: input generator, input constraints, and checkers. I also show that while it might incur some overhead, it is often feasible to design test setups that can work with both formal and random testing techniques.

Beyond tools and techniques to automatically generate test inputs and simplify coverage collection, Chapter 7 focuses on simplifying the developer's task after discovering a bug-revealing input. The RTL-Repair tool presents a new repair synthesis technique that takes a buggy hardware design and a failing testbench as input and produces a repair suggestion that changes the hardware description to pass the testbench while minimizing the size of the change. This tool is orders of magnitude faster than prior work – providing repair suggestions in seconds instead of minutes or hours. A new synthesis technique based on BMC allows us to quickly identify repairs, while windowed sampling and concrete execution provide scalability to longer tests and larger designs. Thus, by combining formal and concrete techniques, we arrive at a superior solution.

## The Future of Hardware Development Tool Research

I spent my PhD designing testing and debugging tools for hardware designs as part of a software engineering research group. This has given me a unique perspective compared to working in a traditional electronic design automation group like many others in my field. Nowadays, software engineering researchers have easy access to high-quality open-source compilers that they can extend to implement their ideas. There are large open-source code bases used heavily in industry, which can serve as realistic benchmarks. In the hardware domain, on the other hand, benchmarks and tools are hard to come by.

While large open-source RTL projects do exist, they are largely written by hobbyists or academics. When the industry releases RTL code, it is often out-of-date, lacks development history and backend scripts, and often contains generated code without the scripts to reproduce it. This poses two issues: (1) it is impossible to study industry practices for academic researchers without working at a company and signing non-disclosure agreements (NDAs), and (2) the industry can easily dismiss any insights and benchmark results obtained on open-source RTL code as not applicable to their internal code bases. Fortunately, in recent years, academic chip designs have become more sophisticated, and industrial research labs have started releasing their experimental designs as open-source RTL. However, there is still a large gap to software engineering, where research papers demonstrate how their tool finds bugs in software code that is used by millions of people. For now, in the hardware domain, only black-box reverse engineering papers can have that same amount of impact.

The lack of open-source RTL that large chip companies would consider representative of their codebases leads researchers at industry research labs to instead opt to use only internal designs when evaluating their work. This makes it impossible to compare results across papers from different companies and academic researchers. It also prevents new investigations of old results. This problem could be tackled if EDA and hardware verification conferences started to encourage and eventually require reproducible software artifacts that include benchmarks and the source code of the proposed tool. This way, researchers from the industry will either have to push their employers to release more RTL code, or they will have to bless existing open-source RTL code by using it in their work.

Beyond a lack of common, agreed-upon benchmarks, hardware design tool research also suffers from a lack of open-source tools that can easily be modified to prototype new research ideas. The first problem is missing support for more sophisticated SystemVerilog language features in open-source tools, restricting the set of benchmarks – particularly from the industry – on which the tool can be evaluated. But even if the language support was fixed, we would still face the problem that often open-source tools like various model checker or simulator implementations severely lack behind their commercial counterparts, making it hard to evaluate whether an improvement to the open-source tools actually advances the state of the art. While we could benchmark the commercial tools to get at least an end-to-end performance comparison, publishing these results is generally prohibited by the EDA software companies in their end-user license agreements.

Countless research prototypes use the open-source synthesis tool yosys to implement new ideas, demonstrating how important solid open-source tools are to research progress. However, yosys is still far from supporting the full SystemVeriog language. In software engineering, on the other hand, researchers who want their tools to work with most C++ code bases can take advantage of Clang and LLVM. We can avoid this problem by building on new, simpler hardware languages with open-source compilers like Chisel, as demonstrated in this thesis. However, the obvious downside is that this restricts us to benchmarks expressed in these new languages, making comparisons to existing work impossible. The new open-source Slang SystemVeriog parser might alleviate some of these problems. While it is not a full-blown compiler, it has shown promising results regarding SystemVerilog parsing capabilities.

While some open-source RTL simulators like Veriator [131] can keep up with commercial tools when it comes to execution speed, open-source formal verification tools are generally thought to be inferior compared to industry tools like Jasper Gold or IBM's SixthSense. While software engineering researchers can often build upon mature open-source tools used in industry, research on formal and semi-formal methods for hardware verification is often reduced to re-inventing the basics. This is a big opportunity for a systems research group to build and maintain a baseline formal verification tool for hardware. While the basic algorithms and ideas are all known and published, combining them in a tool that is scalable and easy to extend for other research groups poses new and interesting systems engineering challenges.

The hardware development tools research community suffers because too much knowledge is locked away as tools and benchmarks inside highly secretive chip companies. Some researchers will have to step up and perform the often-underappreciated work of defining new common benchmark sets that are accepted by both academia and industry and building the open-source tools necessary to quickly prototype new ideas without wasting time re-inventing the wheel every time.

Fortunately, the work on new open-source hardware tools over the last decade has brought us much closer to fulfilling this vision of high-quality common RTL benchmark sets and verification tools. While there is still more work to be done, the tools developed for this thesis demonstrate how far we have come. Without the groundwork of the community, none of these research prototypes would have been feasible to develop. I deeply believe that open-source is the only way forward for our research community, helping to ground our research and ensure its relevance for years to come.

# Bibliography

[1] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. "Fixing Hardware Security Bugs with Large Language Models". In: *arXiv preprint arXiv:2302.01215* (2023).

[2] Hammad Ahmad, Yu Huang, and Westley Weimer. "CirFix: Automatically Repairing Defects in Hardware Design Code". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2022, pp. 990–1003.

[3] Tutu Ajayi, Vidya A Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, et al. "Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–4.

[4] Bijan Alizadeh and Masoud Shiroei. "Automatic Correction of RTL Designs Using a Lightweight Partial High Level Synthesis". In: *Integration* 91 (2023), pp. 173–181.

[5] 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". In: *IEEE Micro* 40 (2020).

[6] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Benjamin Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. 2016. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[7] 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: *DAC Design Automation Conference 2012*. 2012.

[8] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6.* Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2017.

[9] Clark Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability.* 2008.

[10] Scott Beamer and David Donofrio. "Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation". In: *57th ACM/IEEE Design Automation Conference (DAC).* 2020.

[11] Kent Beck. *Test driven development: By example.* Addison-Wesley Professional, 2022.

[12] David Thomas Biancolin. *Automated, FPGA-Based Hardware Emulation of Dynamic Frequency Scaling.* University of California, Berkeley, 2021.

[13] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu, et al. "Bounded Model Checking". In: *Advances in computers* 58.11 (2003), pp. 117–148.

[14] Armin Biere, Keijo Heljanko, and Siert Wieringa. "AIGER 1.9 and beyond". In: (2011).

[15] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. "$\nu$z-an optimizing SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21.* Springer. 2015, pp. 194–199.

[16] Roderick Bloem and Franz Wotawa. "Verification and Fault Localization in VHDL Programs". In: *Journal of the Telematics Engineering Society (TIV)* 2 (2002), pp. 30–33.

[17] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based greybox fuzzing as markov chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* 2016, pp. 1032–1043.

[18] Mark Bohr. "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper". In: *IEEE Solid-State Circuits Society Newsletter* 12 (2007). DOI: `10.1109/N-SSC.2007.4785534`.

[19] Mrinal Bose, Jongshin Shin, Elizabeth M Rudnick, Todd Dukes, and Magdy Abadir. "A genetic approach to automatic bias generation for biased random instruction generation". In: *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on.* Vol. 1. IEEE. 2001, pp. 442–448.

[20] Sébastien Bourdeauducq. *migen.* `https://m-labs.hk/gateware/migen`. 2011.

[21] Frederick P Brooks. "No Silver Bullet: Essence and Accident of Software Engineering". In: (1986).

[22] Niklas Bruns, Vladimir Herdt, Daniel Große, and Rolf Drechsler. “Efficient Cross-Level Processor Verification using Coverage-guided Fuzzing”. In: *Proceedings of the Great Lakes Symposium on VLSI 2022*. 2022.

[23] Randal E Bryant. “Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams”. In: *ACM Computing Surveys (CSUR)* 24 (1992).

[24] 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: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2021, pp. 529–534.

[25] Sadullah Canakci, Chathura Rajapaksha, Leila Delshadtehrani, Anoop Nataraja, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. “ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance”. In: *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE. 2023, pp. 1–12.

[26] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. “Angelic Debugging”. In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011, pp. 121–130.

[27] Kai-hui Chang, Ilya Wagner, Valeria Bertacco, and Igor L Markov. “Automatic Error Diagnosis and Correction for RTL Designs”. In: *2007 IEEE International High Level Design Validation and Test Workshop*. IEEE. 2007, pp. 65–72.

[28] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. “HyPFuzz: Formal-Assisted Processor Fuzzing”. In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023.

[29] Joonwon Choi, Adam Chlipala, and Arvind. “Hemiola: A DSL and Verification Tools to Guide Design and Proof of Hierarchical Cache-Coherence Protocolss”. In: *International Conference on Computer Aided Verification*. Springer. 2022, pp. 317–339.

[30] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. “Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification”. In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017), pp. 1–30.

[31] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. “NuSMV: a new symbolic model checker”. In: *International Journal on Software Tools for Technology Transfer* (2000).

[32] Edmund Clarke and Daniel Kroening. “Hardware Verification Using ANSI-C Programs as a Reference”. In: *ASP-DAC’03*. IEEE. 2003.

[33] Intel Corporation. *Intel (R) C/C++ Compilers Complete Adoption of LLVM*. `https://www.intel.com/content/www/us/en/developer/articles/technical/adoption-of-llvm-complete-icx.html`. Accessed May, 2024.

[34] Intel Corporation. *Moore's Law*. `https://www.intel.com/content/www/us/en/newsroom/resources/moores-law.html`. Accessed May, 2024.

[35] International Business Machines Corporation. *A new Clang-based Front End*. `https://www.ibm.com/docs/en/xl-c-and-cpp-aix/16.1?topic=new-clang-based-front-end`. Accessed May, 2024.

[36] Matthias Cosler, Frederik Schmitt, Christopher Hahn, and Bernd Finkbeiner. "Iterative Circuit Repair Against Formal Specifications". In: *arXiv preprint arXiv:2303.01158* (2023).

[37] Clifford E Cummings et al. "Nonblocking assignments in verilog synthesis, coding styles that kill!" In: *SNUG (Synopsys Users Group) 2000 User Papers* (2000).

[38] John Darringer, Evan Davidson, David J Hathaway, Bernd Koenemann, Mark Lavin, Joseph K Morrell, Khalid Rahmat, Wolfgang Roesner, Erich Schanzenbach, Gustavo Tellez, et al. "EDA in IBM: Past, Present, and Future". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19.12 (2000).

[39] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[40] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9 (1974). DOI: `10.1109/JSSC.1974.1050511`.

[41] Amelia Dobis. "Formal Verification of Hardware using MLIR". MA thesis. ETH Zurich, 2024.

[42] Amelia Dobis, Kevin Laeufer, Hans Jakob Damsgaard, Tjark Petersen, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. "Verification of Chisel Hardware Designs with ChiselVerify". In: *Microprocessors and Microsystems* 96 (2023), p. 104737.

[43] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. "MLIR as Hardware Compiler Infrastructure". In: *Workshop on Open-Source EDA Technology (WOSET)*. 2021.

[44] Mahyar Emami, Thomas Bourgeat, and James Larus. "Parendi: Thousand-Way Parallel RTL Simulation". In: *arXiv preprint arXiv:2403.04714* (2024).

[45] Mahyar Emami, Sahand Kashani, Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, and James R Larus. "Manticore: Hardware-accelerated RTL simulation with static bulk-synchronous parallelism". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 2023, pp. 219–237.

[46] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. "Integration Verification across Software and Hardware for a Simple Embedded System". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 2021, pp. 604–619.

[47] Augie Eriksson and Maanuj Vora. "A Java Backend for ESSENT". In: *Workshop on Open-Source EDA Technology (WOSET).* 2022. URL: `https://woset-workshop.github.io/WOSET2022.html#article-11`.

[48] Brandon Fajardo, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. "RTLFUZZLAB: Building A Modular Open-Source Hardware Fuzzing Framework". In: *Workshop on Open-Source EDA Technology (WOSET).* 2021. URL: `https://woset-workshop.github.io/WOSET2021.html#article-10`.

[49] Shai Fine and Avi Ziv. "Coverage directed test generation for functional verification using bayesian networks". In: *Proceedings of the 40th annual Design Automation Conference.* ACM. 2003, pp. 286–291.

[50] Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon Davidmann. "Verilog HDL and Its Ancestors and Descendants". In: *Proceedings of the ACM on Programming Languages* 4.HOPL (2020), pp. 1–90.

[51] Jeremy Gibbons and Nicolas Wu. "Folding domain-specific languages: deep and shallow embeddings (functional pearl)". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming.* 2014, pp. 339–347.

[52] Tristan Gingold et al. *GHDL - VHDL 2008/93/87 simulator.* `http://ghdl.free.fr/`. 2023. URL: `http://ghdl.free.fr/`.

[53] GitHub. *GitHub Copilot.* `https://copilot.github.com/`. 2023. URL: `https://copilot.github.com/`.

[54] Aman Goel and Karem Sakallah. "Avr: Abstractly verifying reachability". In: *Tools and Algorithms for the Construction and Analysis of Systems* 12078 (2020), p. 413.

[55] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. "Automated Program Repair". In: *Communications of the ACM* 62.12 (2019), pp. 56–65.

[56] Wilson Research Group. *2022 Wilson Research Group Functional Verification Study.* `https://blogs.sw.siemens.com/verificationhorizons/2022/10/10/prologue-the-2022-wilson-research-group-functional-verification-study/`. 2022.

[57] Scott Hazelhurst and Carl-Johan H Seger. "Symbolic trajectory evaluation". In: *Formal Hardware Verification.* Springer, 1997, pp. 3–78.

[58] Matthew Hicks, Cynthia Sturton, Samuel T King, and Jonathan M Smith. "SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems.* 2015.

[59] Chris Higgs, Stuart Hodgson, and Eric Wieser. *cocotb*. `https://github.com/cocotb/cocotb`. 2021.

[60] 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)*. 2021.

[61] "IEC/IEEE International Standard - Verilog(R) Register Transfer Level Synthesis". In: *IEEE/IEC 62142* (2005).

[62] "IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language". In: *IEEE Std. 1800* (2017).

[63] "IEEE Standard for Universal Verification Methodology Language Reference Manual". In: *IEEE Std. 1800.2* (2020).

[64] "IEEE Standard for VHDL Language Reference Manual". In: *IEEE Std. 1076* (2019).

[65] Charalambos Ioannides, Geoff Barrett, and Kerstin Eder. "Feedback-based coverage directed test generation: An industrial evaluation". In: *Haifa Verification Conference*. Springer. 2010, pp. 112–128.

[66] Adam Izraelevitz. *Unlocking Design Reuse with Hardware Compiler Frameworks*. University of California, Berkeley, 2019.

[67] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. "Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations". In: *Proceedings of the 36th International Conference on Computer-Aided Design*. ICCAD '17. 2017.

[68] Shunning Jiang, Yanghui Ou, Peitian Pan, Kaishuo Cheng, Yixiao Zhang, and Christopher Batten. "PyH2: Using PyMTL3 to Create Productive and Open-Source Hardware Testing Methodologies". In: *IEEE Design & Test* 38.2 (2020).

[69] Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten. "Pymtl3: A python framework for open-source hardware modeling, generation, simulation, and verification". In: *IEEE Micro* 40.4 (2020), pp. 58–66.

[70] Tai-Ying Jiang, C-NJ Liu, and Jing Ya Jou. "Estimating Likelihood of Correctness for Error Candidates to Assist Debugging Faulty HDL Designs". In: *2005 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2005, pp. 5682–5685.

[71] Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. "Effective Processor Verification with Logic Fuzzer Enhanced Co-simulation". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 2021, pp. 667–678.

[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)*. 2022.

[73] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*.

[74] Donggyu Kim. *FPGA-Accelerated Evaluation and Verification of RTL Designs*. University of California, Berkeley, 2019.

[75] Donggyu Kim. *risc-v mini*. `https://github.com/ucb-bar/riscv-mini`. 2019.

[76] Donggyu Kim, Adam Izraelevitz, Christopher Celio, Hokeun Kim, Brian Zimmer, Yunsup Lee, Jonathan Bachrach, and Krste Asanović. "Strober: fast and accurate sample-based energy simulation for arbitrary RTL". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. Vol. 44. 3. IEEE Press. 2016, pp. 128–139.

[77] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K Ganai. "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21.12 (2002), pp. 1377–1394.

[78] Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. "Open-Source Formal Verification for Chisel". In: *Workshop on Open-Source EDA Technology (WOSET)*. 2021. URL: `https://woset-workshop.github.io/WOSET2021.html#article-3`.

[79] Kevin Laeufer, Brandon Fajardo, Abhik Ahuja, Vighnesh Iyer, Borivoje Nikolić, and Koushik Sen. "RTL-Repair: Fast Symbolic Repair of Hardware Design Code". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2024.

[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*. 2023.

[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'18)*. 2018.

[82] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pp. 2–14.

[83] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. "GenProg: A Generic Method for Automatic Software Repair". In: *Ieee transactions on software engineering* 38.1 (2011), pp. 54–72.

[84] Caroline Lemieux and Koushik Sen. "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage". In: *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 2018, pp. 475–485.

[85] Tun Li, Hongji Zou, Dan Luo, and Wanxia Qu. "Symbolic Simulation Enhanced Coverage-Directed Fuzz Testing of RTL Design". In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2021.

[86] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Brucek Khailany, Shih-Hsin Wang, and Tsung-Wei Huang. "GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs". In: *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2023, pp. 1–6.

[87] Richard Lin and Kevin Laeufer. *ChiselTest*. `https://github.com/ucb-bar/chiseltest`. 2024.

[88] Derek Lockhart, Gary Zibrat, and Christopher Batten. "PyMTL: A unified framework for vertically integrated computer architecture research". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2014, pp. 280–292.

[89] Fan Long and Martin Rinard. "Automatic Patch Generation by Learning Correct Code". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016, pp. 298–312.

[90] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, and Baris Kasikci. "Debugging in the Brave New World of Reconfigurable Hardware". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2022, pp. 946–962.

[91] David R MacIver, Zac Hatfield-Dodds, et al. "Hypothesis: A new approach to property-based testing". In: *Journal of Open Source Software* 4.43 (2019), p. 1891.

[92] Jean Christophe Madre, Olivier Coudert, and Jean Paul Billon. "Automating the Diagnosis and the Rectification of Design Errors with PRIAM". In: *The Best of ICCAD: 20 Years of Excellence in Computer-Aided Design*. Springer, 1989, pp. 17–27.

[93] Albert Magyar, David Biancolin, John Koenig, Sanjit Seshia, Jonathan Bachrach, and Krste Asanović. "Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ICCAD'19. 2019.

[94] Albert Forte Magyar. *Improving FPGA Simulation Capacity with Automatic Resource Multi-Threading*. University of California, Berkeley, 2021.

[95] Makai Mann and Clark Barrett. “Partial Order Reduction for Deep Bug Finding in Synchronous Hardware”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2020.

[96] Makai Mann, Ahmed Irfan, Florian Lonsing, Yahan Yang, Hongce Zhang, Kristopher Brown, Aarti Gupta, and Clark Barrett. “Pono: a flexible and extensible SMT-based model checker”. In: *International Conference on Computer Aided Verification*. Springer. 2021, pp. 461–474.

[97] Chick Markley. *treadle*. `https://github.com/chipsalliance/treadle`. 2021.

[98] Joao Marques-Silva, Inês Lynce, and Sharad Malik. “Conflict-Driven Clause Learning SAT Solvers”. In: *Handbook of Satisfiability*. 2021.

[99] Kenneth L McMillan. “The SMV System”. In: *Symbolic Model Checking*. 1993.

[100] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. “Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis”. In: *Proceedings of the 38th international conference on software engineering*. 2016, pp. 691–701.

[101] *Migrate from ARM C/C++ Compiler 5 to ARM Compiler 6*. KAN298. ARM. 2017.

[102] Don Mills and Clifford E Cummings. “RTL Coding Styles That Yield Simulation and Synthesis Mismatches”. In: *SNUG (Synopsys Users Group) 1999 Proceedings*. 1999.

[103] Alan Mishchenko et al. “ABC: A System for Sequential Synthesis and Verification”. In: *URL http://www.eecs.berkeley.edu/alanmi/abc* (2007).

[104] Sujit Kumar Muduli, Gourav Takhar, and Pramod Subramanyan. “Hyperfuzzing for SoC security validation”. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. 2020.

[105] A. Nahir, A. Ziv, and S. Panda. “Optimizing test-generation to the execution platform”. In: *17th Asia and South Pacific Design Automation Conference*. 2012.

[106] Gilly Nativ, S Mittennaier, Shmuel Ur, and Avi Ziv. “Cost evaluation of coverage directed test generation for the IBM mainframe”. In: *Test Conference, 2001. Proceedings. International*. IEEE. 2001, pp. 793–802.

[107] Matthew Naylor and Simon Moore. “A generic synthesisable test bench”. In: *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*. IEEE. 2015, pp. 128–137.

[108] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. “Semfix: Program Repair via Semantic Analysis”. In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 772–781.

[109] Aina Niemetz and Mathias Preiner. “Bitwuzla”. In: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*. Springer, 2023. DOI: `10.1007/978-3-031-37703-7\_1`.

[110] Aina Niemetz, Mathias Preiner, and Armin Biere. “Boolector 2.0 system description”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2014).

[111] Aina Niemetz, Mathias Preiner, Claire Wolf, and Armin Biere. “Btor2, BtorMC and Boolector 3.0”. In: *International Conference on Computer Aided Verification*. Springer. 2018, pp. 587–595.

[112] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. “A Compiler Infrastructure for Accelerator Generators”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’21)*. 2021.

[113] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[114] Sam Owre, Sreeranga Rajan, John M Rushby, Natarajan Shankar, and Mandayam Srivas. “PVS: Combining specification, proof checking, and model checking”. In: *International Conference on Computer Aided Verification*. Springer. 1996, pp. 411–414.

[115] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. “Semantic fuzzing with zest”. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 329–340.

[116] Charles Papon et al. *SpinalHDL*. `https://github.com/SpinalHDL/SpinalHDL`. 2024.

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

[118] Mathias Preiner and Armin Biere. “Hardware Model Checking Competition 2019”. In: (2019).

[119] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. “An Analysis of Patch Plausibility and Correctness for Generate-And-Validate Patch Generation Systems”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 2015, pp. 24–36.

[120] Jiann-Chyi Ran, Yi-Yuan Chang, and Chia-Hung Lin. “An Efficient Mechanism for Debugging RTL Description”. In: *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, 2003. Proceedings*. IEEE. 2003, pp. 370–373.

[121] Anthon Vincent Riber. “Power-efficient hardware platform for Spiking neural network”. MA thesis. Department of Applied Mathematics and Computer Science, Technical University of Denmark, 2020.

[122] “RISCV-DV”. In: (). URL: `https://github.com/google/riscv-dv`.

[123] Katharina Ruep and Daniel Große. “SpinalFuzz: Coverage-Guided Fuzzing for SpinalHDL Designs”. In: *2022 IEEE European Test Symposium (ETS)*. IEEE. 2022.

[124] Sameer D Sahasrabuddhe, Hakim Raja, Kavi Arya, and Madhav P Desai. "AHIR: A Hardware Intermediate Representation for Hardware Generation from High-level Programs". In: *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*. 2007.

[125] Alberto Sangiovanni-Vincentelli. "The Tides of EDA". In: *IEEE Design & Test of Computers* 20.6 (2003), pp. 59–75.

[126] Priscila Santiesteban, Yu Huang, Westley Weimer, and Hammad Ahmad. "CirFix: Automated Hardware Repair and its Real-World Applications". In: *IEEE Transactions on Software Engineering* (2023).

[127] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. "LLHD: A Multi-level Intermediate Representation for Hardware Description Languages". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. 2020.

[128] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "{AddressSanitizer}: A fast address sanity checker". In: *2012 USENIX annual technical conference (USENIX ATC 12)*. 2012, pp. 309–318.

[129] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. "$\mu$IR - An intermediate representation for transforming and optimizing the microarchitecture of application accelerators". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. 2019.

[130] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. "Checking safety properties using induction and a SAT-solver". In: *International conference on formal methods in computer-aided design*. Springer. 2000, pp. 127–144.

[131] Wilson Snyder et al. *Verilator*. `https://www.veripool.org/wiki/verilator`. 2023. URL: `https://www.veripool.org/wiki/verilator`.

[132] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. "Cascade: CPU Fuzzing via Intricate Program Generation". In: *Proc. 33rd USENIX Secur. Symp.* 2024.

[133] Giovanni Squillero. "MicroGP - an evolutionary assembly program generator". In: *Genetic Programming and Evolvable Machines* 6.3 (2005), pp. 247–263.

[134] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "Driller: Augmenting fuzzing through selective symbolic execution." In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

[135] Stuart Sutherland and Don Mills. "Synthesizing systemverilog busting the myth that systemverilog is only for verification". In: *SNUG Silicon Valley* (2013), p. 24.

[136] Synopsys. *VCS*. `https://www.synopsys.com/verification/simulation.html`. 2023. URL: `https://www.synopsys.com/verification/simulation.html`.

[137] Shinya Takamaeda-Yamazaki. “Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL”. In: *Applied Reconfigurable Computing: 11th International Symposium, ARC 2015, Bochum, Germany, April 13-17, 2015, Proceedings 11*. Springer. 2015, pp. 451–460.

[138] Serdar Tasiran, Farzan Fallah, David G Chinnery, Scott J Weber, and Kurt Keutzer. “A functional validation technique: biased-random simulation guided by observability-based coverage”. In: *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*. IEEE. 2001, pp. 82–88.

[139] Michael B. Taylor. “Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse”. In: *Proceedings of the 49th Annual Design Automation Conference*. DAC ’12. New York, NY, USA: Association for Computing Machinery, 2012. DOI: `10.1145/2228360.2228567`.

[140] T. N. Theis and H. Wong. “The End of Moore’s Law: A New Beginning for Information Technology”. In: *Computing in Science & Engineering* 19 (2017). DOI: `10.1109/MCSE.2017.29`.

[141] 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)*. 2022, pp. 3237–3254.

[142] Lenny Truong and Pat Hanrahan. “A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity”. In: *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. 2019.

[143] Lenny Truong, Steven Herbst, Rajsekhar Setaluri, Makai Mann, Ross Daly, Keyi Zhang, Caleb Donovick, Daniel Stanley, Mark Horowitz, Clark Barrett, et al. “fault: A Python Embedded Domain-Specific Language for Metaprogramming Portable Hardware Verification Components”. In: *International Conference on Computer Aided Verification*. CAV’20. 2020.

[144] Mike Turpin. “The Dangers of Living with an X (bugs hidden in your Verilog)”. In: *Synopsys Users Group Meeting*. 2003.

[145] Shmuel Ur and Yaov Yadin. “Micro Architecture Coverage Directed Generation of Test Programs”. In: *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*. DAC ’99. New Orleans, Louisiana, USA: ACM, 1999, pp. 175–180. ISBN: 1-58113-109-7.

[146] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. “Modular Deductive Verification of Multiprocessor Hardware Designs”. In: *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II 27*. Springer. 2015, pp. 109–127.

[147] Yakir Vizel and Arie Gurfinkel. “Interpolating property directed reachability”. In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 260–276.

[148] J. Wang, H. Li, T. Lv, T. Wang, X. Li, and S. Kundu. "Abstraction-Guided Simulation Using Markov Analysis for Functional Verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.2 (2016), pp. 285–297.

[149] Tianrui Wei, Kevin Laeufer, Katie Lim, Jerry Zhao, Koushik Sen, Jonathan Balkind, and Krste Asanovic. "Zoomie: A Software-like Debugging Tool for FPGAs". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2024, pp. 1048–1062.

[150] Catherine whitequark. *amaranth*. `https://github.com/amaranth-lang/amaranth`. 2022.

[151] Stephen Williams et al. *Icarus Verilog*. `https://steveicarus.github.io/iverilog/`. 2023. URL: `https://steveicarus.github.io/iverilog/`.

[152] Claire Wolf. *SymbiYosys*. 2021. URL: `https://github.com/YosysHQ/SymbiYosys`.

[153] Claire Wolf and Johann Glaser. "Yosys-a free Verilog synthesis suite". In: *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*. 2013.

[154] Jiang Wu, Zhuo Zhang, Deheng Yang, Xiankai Meng, Jiayu He, Xiaoguang Mao, and Yan Lei. "Fault Localization for Hardware Design Code with Time-Aware Program Spectrum". In: *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE. 2022, pp. 537–544.

[155] Jin Yang and C-JH Seger. "Introduction to generalized symbolic trajectory evaluation". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 11.3 (2003), pp. 345–353.

[156] Michał Zalewski. *American Fuzzy Lop Technical Details*. `http://lcamtuf.coredump.cx/afl/technical_details.txt`. Accessed April, 2018. 2014.

[157] Keyi Zhang, Zain Asgar, and Mark Horowitz. "Bringing Source-Level Debugging Frameworks to Hardware Generators". In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. DAC '22. 2022.

[158] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. "End-to-End Automated Exploit Generation for Validating the Security of Processor Designs". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018.

[159] Rui Zhang, Natalie Stanley, Christopher Griggs, Andrew Chi, and Cynthia Sturton. "Identifying Security Critical Properties for the Dynamic Verification of a Processor". In: ASPLOS '17. 2017.

[160] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine". In: (2020).

# Appendix A

# Chapter 5 Example Source Code

```
class GcdInputBundle(val w: Int) extends Bundle {
  val value1 = UInt(w.W)
  val value2 = UInt(w.W)
}

class GcdOutputBundle(val w: Int) extends Bundle {
  val value1 = UInt(w.W)
  val value2 = UInt(w.W)
  val gcd    = UInt(w.W)
}

class DecoupledGcd(width: Int) extends Module {
  val input = IO(Flipped(Decoupled(new GcdInputBundle(width))))
  val output = IO(Decoupled(new GcdOutputBundle(width)))

  val xInitial    = Reg(UInt())
  val yInitial    = Reg(UInt())
  val x           = Reg(UInt())
  val y           = Reg(UInt())
  val busy        = RegInit(false.B)
  val resultValid = RegInit(false.B)

  input.ready := ! busy
  output.valid := resultValid
  output.bits := DontCare

  when(busy)  {
    when(x > y) {
      x := x - y
    }.otherwise {
      y := y - x
    }
```

```scala
      when(x === 0.U || y === 0.U) {
        when(x === 0.U) {
          output.bits.gcd := y
        }.otherwise {
          output.bits.gcd := x
        }

        output.bits.value1 := xInitial
        output.bits.value2 := yInitial
        resultValid := true.B

        when(output.ready && resultValid) {
          busy := false.B
          resultValid := false.B
        }
      }
    }.otherwise {
      when(input.valid) {
        val bundle = input.deq()
        x := bundle.value1
        y := bundle.value2
        xInitial := bundle.value1
        yInitial := bundle.value2
        busy := true.B
      }
    }
  }

```

Listing 1: Greatest common denominator (GCD) circuit from the Chisel template repository. Released to the public domain by its authors.

```scala
class GCDSpec extends AnyFreeSpec with ChiselScalatestTester {

  "Gcd should calculate proper greatest common denominator" in {
    test(new DecoupledGcd(16)) { dut =>
      dut.input.initSource()
      dut.input.setSourceClock(dut.clock)
      dut.output.initSink()
      dut.output.setSinkClock(dut.clock)

      val testValues = for { x <- 0 to 10; y <- 0 to 10} yield (x, y)
      val inputSeq = testValues.map { case (x, y) =>
```

```
          (new GcdInputBundle(16)).Lit(_.value1 -> x.U, _.value2 -> y.U)
        }
        val resultSeq = testValues.map { case (x, y) =>
          (new GcdOutputBundle(16)).Lit(
            _.value1 -> x.U,
            _.value2 -> y.U,
            _.gcd -> BigInt(x).gcd(BigInt(y)).U
          )
        }

        fork {
          // push inputs into the calculator, stall for 11 cycles
          val (seq1, seq2) = inputSeq.splitAt(resultSeq.length / 3)
          dut.input.enqueueSeq(seq1)
          dut.clock.step(11)
          dut.input.enqueueSeq(seq2)
        }.fork {
          // retrieve computations, stall for 10 cycles
          val (seq1, seq2) = resultSeq.splitAt(resultSeq.length / 2)
          dut.output.expectDequeueSeq(seq1)
          dut.clock.step(10)
          dut.output.expectDequeueSeq(seq2)
        }.join()

      }
    }
  }
```

Listing 2: A ChiselTest-based test for the GCD circuit from Listing 1. Released to the public domain by its authors.