# Simulator Independent Coverage for RTL Hardware Languages


Kevin Laeufer
laeufer@eecs.berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

Vighnesh Iyer
vighnesh.iyer@eecs.berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

David Biancolin[□]
biancolin@eecs.berkeley.edu
SiFive
San Mateo, CA, USA

Jonathan Bachrach[□]
jrb@pobox.com
JITX
Berkeley, CA, USA

Borivoje Nikolić
bora@eecs.berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

Koushik Sen
ksen@eecs.berkeley.edu
University of California, Berkeley
Berkeley, CA, USA



## ABSTRACT

We demonstrate a new approach to implementing automated coverage metrics including line, toggle, and finite state machine coverage. Each metric is implemented through a compiler pass with a report generator. They are decoupled from the backend simulation, emulation, or formal verification tool through a simple API designed around a single new `cover` primitive. Our prototype for the Chisel hardware construction language demonstrates support across three software simulators, the FPGA-accelerated FireSim simulator and a formal tool. We demonstrate collecting line coverage while booting Linux with FireSim at a target frequency of 65 MHz. By construction, coverage can be trivially merged across backends.



## CCS CONCEPTS

• Hardware → Simulation and emulation; Coverage metrics.

## KEYWORDS

RTL, FPGA, FSM Coverage, Line Coverage, Toggle Coverage, Hardware Compiler

ACM Reference Format:
Kevin Laeufer, Vighnesh Iyer, David Biancolin, Jonathan Bachrach, Borivoje Nikolić, and Koushik Sen. 2023. Simulator Independent Coverage for RTL Hardware Languages. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3582016.3582019


## 1 INTRODUCTION

Dynamic verification is the major workhorse for pre-silicon verification of digital circuit designs. As soon as the register transfer level (RTL) description of a circuit is written, it can be simulated with one of many open-source or commercial simulators. In order to simulate the environment in which the circuit is supposed to operate, designers write testbenches either in their design language or in a variety of unit testing frameworks like fault [28], cocotb [10] or chiseltest [7, 16]. Besides testing individual modules, dynamic verification is also used for system-level or integration tests in which a complete System on Chip (SoC) is simulated. Since the execution speed of software simulators degrades with larger designs, emulation or FPGA-accelerated simulation platforms are often used [13].

Dynamic verification is most effective when it has exercised the full functionality of the design. Coverage metrics [20] are an approximation of the input stimuli' effectiveness in exercising the targeted design. Verification engineers craft stimuli to hit an increasing number of coverpoints in the design and thus gain confidence in the thoroughness of their test suite.

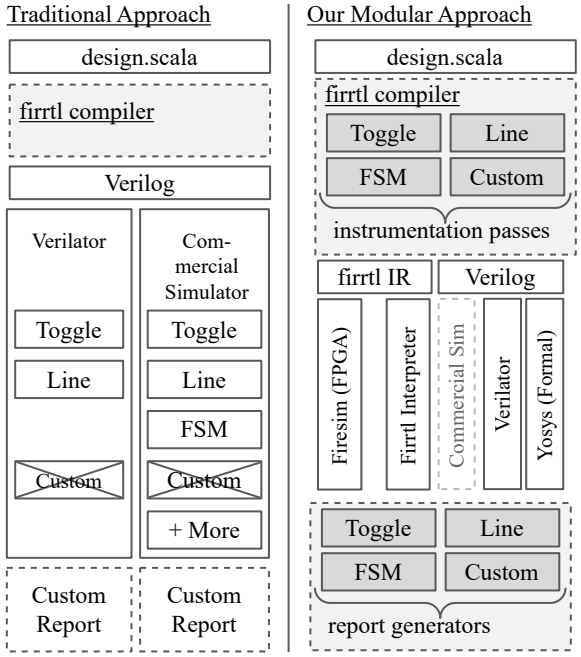


Figure 1: Traditionally, automated coverage collection is part of a monolithic simulator. Users are limited to the coverage metrics that the simulator authors have chosen to provide. We instead implement every coverage metric as a single in-strumentation pass in the firrtl compiler and a simulator-
independent 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.

[□]Afiliated with University of California, Berkeley at the time of this work.



ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada



ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582019

There are several issues with the status quo of coverage instrumentation and collection that we aim to address:

(1) Most open-source or innovative research simulators lack support for collecting and reporting automated coverage metrics [6, 13, 18, 25].
(2) For tools that do support these metrics, their custom implementation makes merging coverage across various software or FPGA-accelerated simulators and formal tools dificult.
(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 paper, we present a 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 1 provides an overview of our system.

We implemented our approach for the Chisel hardware construction language and the FIRRTL compiler [5, 12]. 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 the implementation of new coverage metrics can be challenging we found that adding support for new simulators was fairly simple. Besides Verilator, we also provide support for a FIRRTL simulator called treadle, for the ESSENT simulator [6], the FPGA-accelerated FireSim simulator [13] as well as a formal tool for trace generation.

# 2 BACKGROUND

## 2.1 Coverage

Tests of RTL designs generally involve test stimuli that exercise the design as well as checks that catch when the design is behaving incorrectly, e.g., assertions or comparison of outputs to a golden model. In order to measure whether the inputs cover all interesting behaviors of the design, various coverage metrics have been proposed [20].

Simple structural or code coverage is based on the idea that if the designer writes a statement in the RTL language it has to serve a purpose and thus should be executed at least once. Toggle coverage follows a similar thought pattern: If a wire in the circuit is always stuck at one or zero, then either it should not be there or the design has not been thoroughly tested.

In order to gain insight into more high-level design functionality engineers often manually annotate functional coverage that describes the different scenarios that a circuit was designed for, e.g., one wants to see a processor pipeline resolve a read-after-write hazard. While most functional coverpoints are user-defined, some can also be automatically generated through knowledge of the design patterns that RTL engineers use. The prime example of this is the finite state machine (FSM) coverage for which a simulator extracts all states and possible transitions and counts how often each is covered.

## 2.2 Hardware Construction Languages

With ever-increasing SoC complexity, many designers aim to write RTL generators that can be extensively parameterized and therefore reused. A prime example of this is the RocketChip SoC generator which essentially takes in a list of devices to instantiate (e.g. cores, peripherals, accelerators) and automatically creates device and in-terconnect RTL [4].

Hardware construction languages (HCLs) implement the generator concept in a general-purpose programming language such as Scala [5] or Python [27, 29]. They provide RTL primitives as a library of objects and allow the designer to write Scala or Python programs that connect the RTL constructs into a final design.

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 as if 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.

## 2.3 RTL Intermediate Representations

Many of the new hardware construction languages define an intermediate representation (IR) that is used to lower higher-level language constructs to structural Verilog [12, 27, 29]. Besides translating the custom constructs to a small subset of structural Verilog that is compatible with commercial and open-source backend tools, the compiler infrastructure can also be used to add additional features to the circuit.

In this paper, we write FIRRTL compiler passes to instrument the IR with automated coverage metrics. There has been recent work on new open-source compilers and intermediate representations for high-level synthesis [19, 22, 24], as well as for established industry languages like Verilog [23, 31], and the circt project which tries to be a unifying compiler framework for hardware construction [8]. Most of these compilers also emit structural Verilog which can then be simulated. Our approach to automated coverage and the simulator interface we propose is also applicable to these new languages and frameworks.

## 2.4 Lowering to Structural Verilog

Traditional HDLs like VHDL and Verilog include synthesizable constructs, which can be mapped to hardware and non-synthesizable

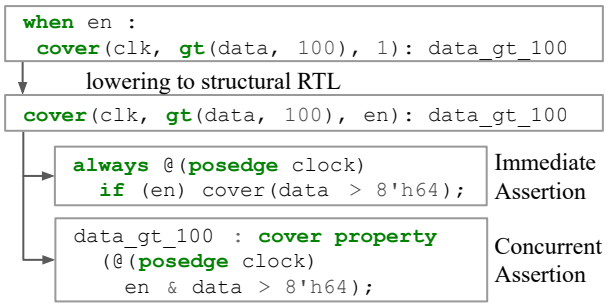


Figure 2: After lowering the **cover** statement to structural RTL, it can be emitted to SystemVerilog as an immediate or concurrent assertion.

constructs that are mostly used for testing. While all of these features can be very convenient for testing circuits, they come with high implementation complexity. So far there is not a single open-source simulator for SystemVerilog that is fully standard compliant [1].

Most new hardware construction languages thus try to emit much simpler, structural Verilog that can be understood by a wide range of tools. More complex features are instead integrated into the frontend language and lowered by the compiler into simpler Verilog features that result in the same behavior. The common subset that is generally well supported by a wide range of tools is the synthesizable subset of the Verilog standard [1].

## 2.5 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 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 [2], 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, which 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.

[1]The results of a SystemVerilog compliance test for many open-source tools can be found at https://symbiflow.github.io/sv-tests-results

[2]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.

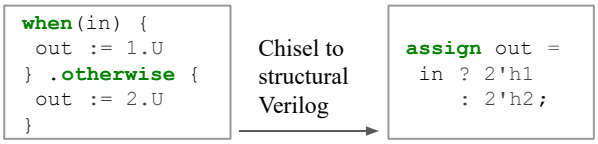


Figure 3: In this example, the translation to structural Verilog replaces a branch with a conditional assignment. Therefore, 100% line coverage on the generated Verilog does not necessarily imply complete line coverage of the Chisel source.

# 3 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, 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 that it is declared in. This way simulators can report coverage results as a simple map from the name of the cover statement (including its path in the module instance hierarchy) to a non-negative integer that represents 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.

## 3.1 Treadle

Treadle [18] 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 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.

## 3.2 Verilator

Verilator [25] 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.

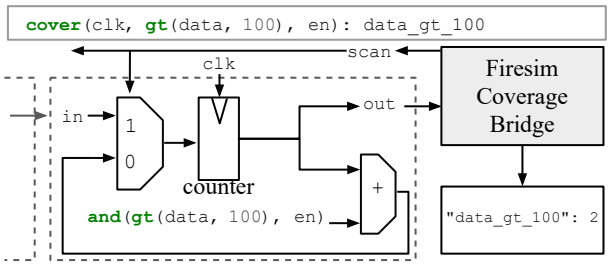


Figure 4: We generate saturating counters and a scan chain for all **cover** statements for FPGA-accelerated simulation with FireSim.

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 2. By default, we generate immediate cover statements [2] as those are the only form supported by the open-source Yosys [31] synthesis tool covered in Section 3.4.

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.

## 3.3 FireSim

FireSim [13] is an open-source, cycle-accurate FPGA-accelerated RTL simulator, which at its core, uses a custom compiler based on FIRRTL [17] 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, as well as FPGA-optimized multi-cycle 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 4). The bit-width of the counter is a parameter set by the user in order 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, freezing 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.

Table 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 |

## 3.4 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 come up with a proof 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 [30]. 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.

## 3.5 ESSENT

ESSENT is a high-performance simulator prototype [6] with little debugging support. In order to gain a sense of how hard it would be to add support for a fifth tool, after the basic idea had been validated with the other four backends, we recorded the time spent. 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.

# 4 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. 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 1 contains an overview of all coverage metrics implemented for this paper, 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. There are many potential improvements that could be made in order to generate interactive HTML reports, or similar, which would significantly increase the amount of code in the report generators.

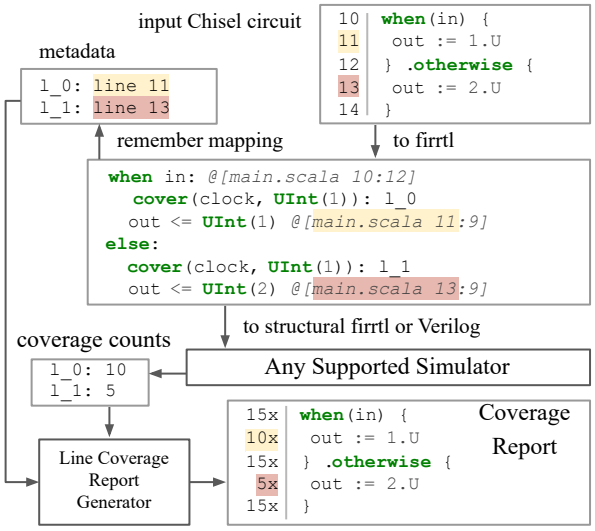


Figure 5: The line coverage pass instruments every **when** statement in the FIRRTL circuit. The mapping from lines to branches is used to generate the coverage report from the counts reported by the simulator.

## 4.1 Branch and Line Coverage

Branch coverage counts how often a branch is taken in the HDL or HCL source code. From this information line and statement coverage can be derived 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 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 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 5.

In order to turn the branch coverage information into actual line coverage, additional information is needed. We scan all statements that are 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 that are covered by it. 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.

## 4.2 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.

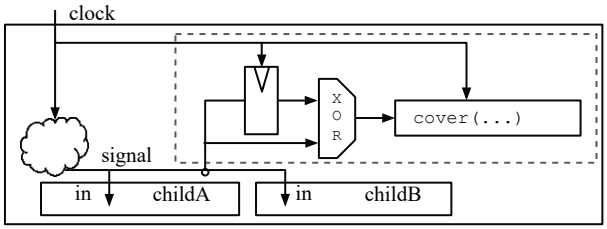


Figure 6: The toggle coverage pass adds a register and a xor gate. It avoids redundant instrumentation for signals that always have the same value.

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 6 the "signal" wire in the top module always carries the same value as the "in" ports of the two child modules. The alias information is used by our toggle coverage pass to only instrument a single signal from each alias group. An important example is the global reset signal, which only gets instrumented 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.

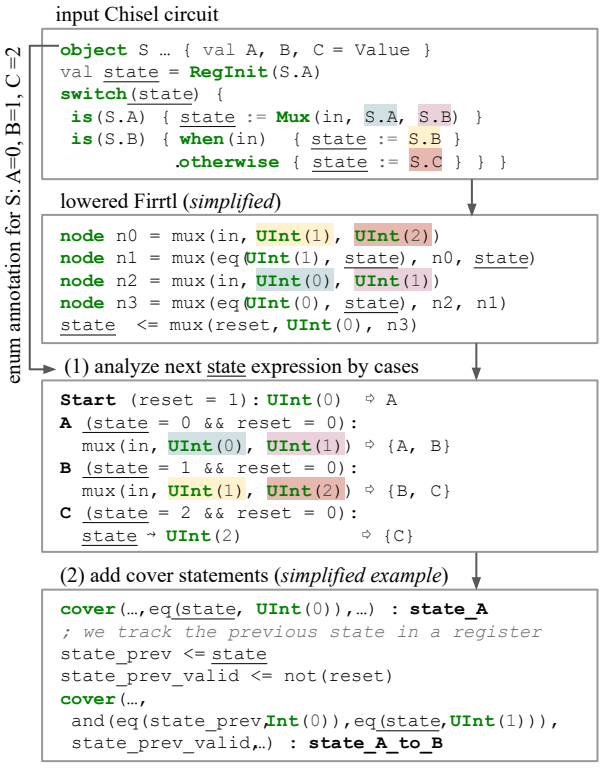


Figure 7: Finite state machine (FSM) coverage assumes that the state register uses a ChiselEnum. We first analyze all possible next states by simplifying the state update expression for each possible current state. We then add **cover** statements for all states and possible transitions.

Table 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 [14] | 126,550 | 3.34 s | 157 | 4,042 |
| TLRAM [4] | 816,473 | 1.45 s | 8 | 2,532 |
| serv-chisel | 828,931 | 1.05 s | 79 | 725 |
| NeuroProc [21] | 53,455,204 | 40.38 s | 809 | 4,786 |

## 4.3 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 registers that contain values from a **ChiselEnum**. The annotation also tells us all legal states that were defined as part of the enum. Figure 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 [4] 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.

## 4.4 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.

# 5 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.

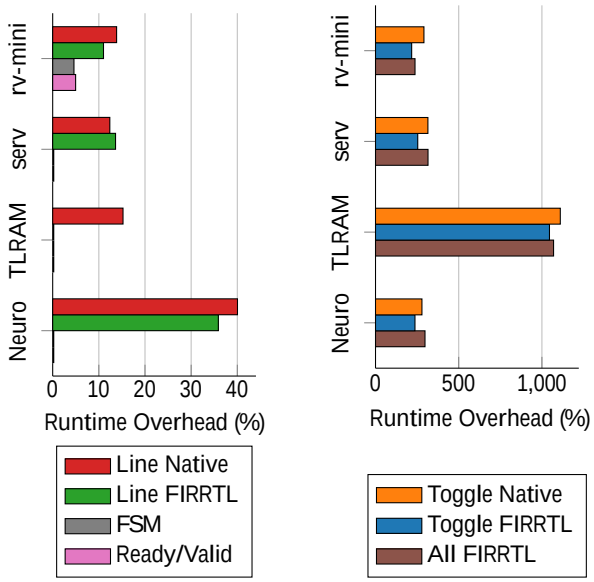


Figure 8: Coverage instrumentation overhead on Verilator v4.034. For TLRAM, the measured overhead of our FIRRTL line coverage is close to zero.

## 5.1 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 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 [3].

Figure 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.

[3] 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.

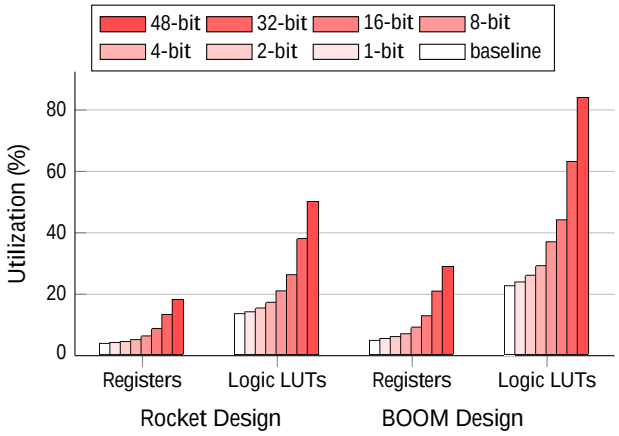


Figure 9: FireSim simulator FPGA resource utilization versus counter width on two different processor designs.

## 5.2 FireSim Coverage Overhead

We applied our line coverage instrumentation to two different SoC designs from the Chipyard framework [3]. The first includes four, in-order scalar Rocket [4] cores and the second uses a single out-of-order BOOM [33] 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 [13]. 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 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 suficient 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 10 illustrates the □□ □ 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 (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.

## 5.3 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.

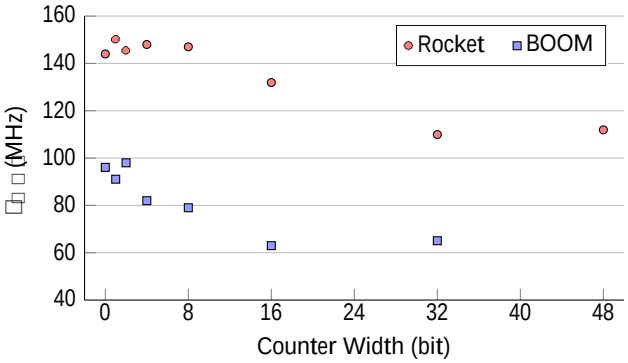


Figure 10: FireSim simulator □□ □ versus counter width. A bit width of zero represents the baseline with no coverage support. Note, the 48 bit BOOM configuration did not place due to resource limitations.

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 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.

## 5.4 Coverage as Fuzzing Feedback

We demonstrate how our approach to coverage can be useful – not only for human developers – but also as automated feedback in a fuzz testing setup. Mutational coverage-directed fuzz testing has recently been applied to RTL designs [11, 15, 26]. However, it is still unclear which feedback metric should be used to drive the input generation. In the past, it has been dificult to switch feedback metrics since they are tied to a particular simulator. However, with the `cover` statement-based approach suggested in this paper, any metric that we implemented an instrumentation pass for can be

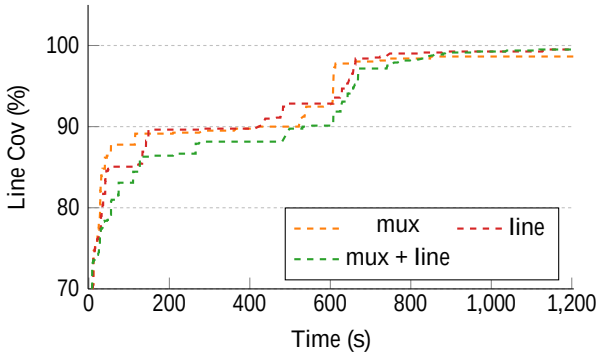


Figure 11: Cumulative line coverage of inputs discovered through fuzzing with various feedback metrics. Averaged over five runs.

used. To demonstrate, we created a simple fuzzing setup, connecting the AFL fuzzer [32] to a rfuzz-style harness [15] using the RTL Fuzz Lab infrastructure [9]. 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 11 shows cumulative line coverage for different feedback metrics.

## 5.5 Formal Trace Generation

As explained in Section 3.4 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.

# 6 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 which we would like to share: In the special case where we have a large number 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 eficiently 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 [11]. Figure 12 demonstrates the exponential blowup when trying to use our `cover` statement and sketches eficient software and hardware implementations of `cover-values` inspired by prior work [11].

# 7 CONCLUSION

Modern hardware verification flows rely on a variety of different simulators, emulators, and formal verification tools. In this paper,

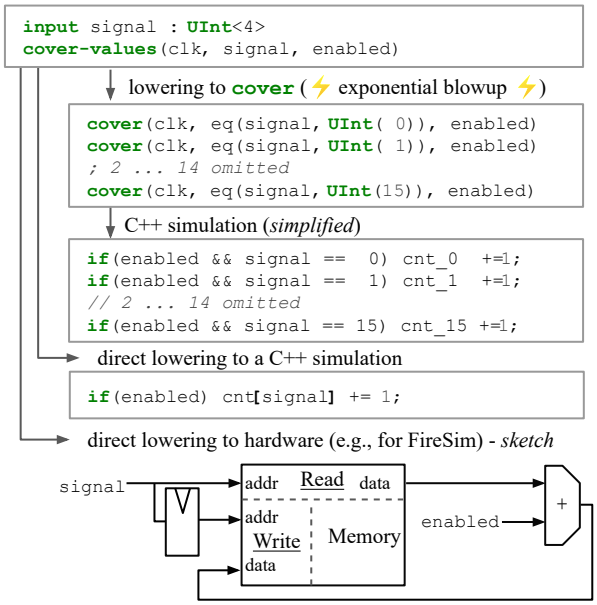


Figure 12: Covering all signal values with the **cover** statement leads to an exponential blowup. A **cover-values** statement could be lowered directly to significantly more eficient software and hardware implementations.

we demonstrate how a compiler-centered approach that lowers automated coverage metrics to a single primitive allows for uniform coverage support across backends. Adding support to a new simulator can be done 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 [25] as well as the FPGA accelerated FireSim simulator which simulates a four-core SoC at 65 MHz effective target frequency, while commercial emulators are generally known to be significantly slower.

Besides broad support for all coverage metrics, our technique enables features that would be dificult to support with a monolithic design where every coverage metric is hard-coded into the simulator: We are able to 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. We are able to re-use any combination of our automated metrics to serve as feedback to a fuzzer for automated input generation.

# ACKNOWLEDGMENTS

We would like to express our gratitude to several members of the Chisel and greater hardware open-source community: Deborah Soung helped us 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 which ultimately lead us to the idea behind this paper. Wilson Snyder wrote the patch to add a "per-instance" coverage feature to Verilator.

This work was supported in part by Semiconductor Research Corporation, by NSF grants CCF-1900968, CCF-1908870, and CNS-1817122 and by SLICE Lab industrial sponsors and affiliates Amazon, Apple, Google, Intel, Qualcomm, and Western Digital, as well as by SKY lab industrial sponsors and affiliates Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Nexla, Samsung SDS, Uber, and VMware. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

# A ARTIFACT APPENDIX

## A.1 Abstract

Our artifact includes the implementation of our coverage passes and report generators, our modifications to FireSim as well as a simple circuit fuzzer. Most results from our paper can be reproduced on a standard x86 Linux computer, however, for the FireSim performance and area/frequency results, a more complicated setup on AWS cloud FPGAs is necessary.

## A.2 Artifact check-list (meta-information)

- Run-time environment: Linux, AWS FPGA Developer AMI
- Hardware: x86 Computer, AWS Cloud FPGAs
- Experiments: Verilator performance overhead, fuzzing, FireSim overhead
- How much disk space required (approximately)?: 2GB (local), 200GB (on AWS)
- How much time is needed to prepare workflow (approximately)?: 1h (local), 1h (on AWS)
- How much time is needed to complete experiments (approximately)?: 5h (local), 5h (on AWS)
- Publicly available?: Yes. On Github: https://github.com/ekiwi/simulator-independent-coverage and https://github.com/ekiwi/firesim
- Code licenses?: 2-Clause BSD and Apache 2.0
- Archived?: https://doi.org/10.5281/zenodo.7592871

## A.3 Description

A.3.1 How to access. We recommend cloning the github repository for the latest code:

https://github.com/ekiwi/simulator-independent-coverage

A.3.2 Software dependencies. Our artifact has been tested on recent Fedora 37 and Ubuntu 20.04 Linux. C++ build tools, bc, sbt, Java (e.g., OpenJDK), hyperfine, and python3 with the scipy and matplotlib libraries need to be installed.

## A.4 Installation

Please follow the instructions in the Readme.md provided as part of the artifact.

## A.5 Evaluation and expected results

The artifact contains scripts to reproduce the following:

- Benchmark statistics in Table 2 (local, 20min)
- Verilator overhead in Figure 8 (local, 2h)
- Fuzzing coverage over time in Figure 11 (local, 2h)
- FireSim resource overhead in Figure 9 and 10 (AWS, 5h)
- Section 5.2 Linux boot times for RocketChip (AWS, 10min)

The Readme.md provided with the artifact contains detailed instructions on how to reproduce each item.

## A.6 Experiment customization

New coverage passes can be added by extending the Scala SBT project in the coverage folder. The provided passes and report generators can serve as a starting point.

New circuits and testbenches to measure overhead on Verilator can be added by modifying the Makefile in the benchmarks folder and copying over the FIRRTL circuit and a C++ testbench.

## A.7 Note

The support for the `cover` statement is part of upstream Treadle since v1.5.0: https://github.com/chipsalliance/treadle

The code to interface with Verilator and convert its custom coverage format into our standard map from `cover` statement name to count is part of upstream ChiselTest since v0.5.0: https://github.com/ucb-bar/chiseltest

Our artifact depends on binary JARs of both Treadle and ChiselTest from the Maven package repository.

Experimental support for `cover` statements in ESSENT can be found on a public fork: https://github.com/ekiwi/essent/tree/coverage

## A.8 Methodology

Submission, reviewing and badging methodology.

# REFERENCES

[1] 2005. IEC/IEEE International Standard - Verilog(R) Register Transfer Level Synthesis. IEEE/IEC 62142 (2005).

[2] 2017. IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language. IEEE Std. 1800 (2017).

[3] 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. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. IEEE Micro 40 (2020). https://doi.org/10.1109/mm.2020.2996616

[4] 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. 2016. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17. University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In DAC Design Automation Conference 2012. https://doi.org/10.1145/2228360.2228584

[6] Scott Beamer and David Donofrio. 2020. Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation. In 57th ACM/IEEE Design Automation Conference (DAC). https://doi.org/10.1109/DAC18072.2020.9218632

[7] Andrew Dobis, Kevin Laeufer, Hans Jakob Damsgaard, Tjark Petersen, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. 2023. Verification of Chisel Hardware Designs with ChiselVerify. Microprocessors and Microsystems 96 (2023). https://doi.org/10.1016/j.micpro.2022.104737

[8] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. 2021. MLIR as Hardware Compiler Infrastructure. In Workshop on Open-Source EDA Technology (WOSET). https://woset-workshop.github.io/WOSET2021.html#article-6

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

[10] Chris Higgs, Stuart Hodgson, and Eric Wieser. 2021. cocotb. https://github.com/cocotb/cocotb.
[11] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. DIFUZZRTL: Differential Fuzz Testing to Find CPU Bugs. In 2021 IEEE Symposium on Security and Privacy (SP). https://doi.org/10.1109/SP40001.2021.00103
[12] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. In Proceedings of the 36th International Conference on Computer-Aided Design (ICCAD '17). https://doi.org/10.1109/ICCAD.2017.8203780
[13] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolić, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2018. 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). https://doi.org/10.1109/ISCA.2018.00014
[14] Donggyu Kim. 2019. risc-v mini. https://github.com/ucb-bar/riscv-mini.
[15] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. In 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18). https://doi.org/10.1145/3240765.3240842
[16] Richard Lin and Kevin Laeufer. 2018 - 2023. ChiselTest. https://github.com/ucb-bar/chiseltest.
[17] Albert Magyar, David Biancolin, John Koenig, Sanjit Seshia, Jonathan Bachrach, and Krste Asanović. 2019. Golden Gate: Bridging The Resource-Eficiency Gap Between ASICs and FPGA Prototypes. In 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) (ICCAD'19). https://doi.org/10.1109/ICCAD45719.2019.8942087
[18] Chick Markley. 2021. treadle. https://github.com/chipsalliance/treadle.
[19] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. 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). https://doi.org/10.1145/3445814.3446712
[20] Andrew Piziali. 2007. Functional Verification Coverage Measurement and Analysis. Springer Science & Business Media.
[21] Anthon Vincent Riber. 2020. Power-eficient Hardware Platform for Spiking Neural Network. Master's thesis. Department of Applied Mathematics and Computer Science, Technical University of Denmark. https://github.com/Thonner/NeuromorphicProcessor
[22] Sameer D Sahasrabuddhe, Hakim Raja, Kavi Arya, and Madhav P Desai. 2007. 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). https://doi.org/10.1109/VLSID.2007.28
[23] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. 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). https://doi.org/10.1145/3385412.3386024
[24] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. 2019. □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). https://doi.org/10.1145/3352460.3358292
[25] Wilson Snyder et al. 2022. Verilator. https://www.veripool.org/wiki/verilator
[26] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2022. Fuzzing Hardware Like Software. In 31st USENIX Security Symposium (USENIX Security 22).
[27] Lenny Truong and Pat Hanrahan. 2019. 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). https://doi.org/10.4230/LIPIcs.SNAPL.2019.7
[28] Lenny Truong, Steven Herbst, Rajsekhar Setaluri, Makai Mann, Ross Daly, Keyi Zhang, Caleb Donovick, Daniel Stanley, Mark Horowitz, Clark Barrett, et al. 2020. fault: A Python Embedded Domain-Specific Language for Metaprogramming Portable Hardware Verification Components. In International Conference on Computer Aided Verification (CAV'20). https://doi.org/10.1007/978-3-030-53288-8_19
[29] whitequark. 2022. amaranth. https://github.com/amaranth-lang/amaranth.
[30] Claire Wolf. 2021. SymbiYosys. https://github.com/YosysHQ/SymbiYosys
[31] Claire Wolf and Johann Glaser. 2013. Yosys - A Free Verilog Synthesis Suite. In Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip).
[32] Michał Zalewski. 2014. American Fuzzy Lop Technical Details. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed April, 2018.
[33] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanović. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In Fourth Workshop on Computer Architecture Research with RISC-V. https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf

Received 2022-10-20; accepted 2023-01-19