Verification of Chisel Hardware Designs with ChiselVerify

Andrew Dobis [a,c,*], Kevin Laeufer [b], Hans Jakob Damsgaard [a,d,1], Tjark Petersen [a], Kasper Juul Hesse Rasmussen [a], Enrico Tolotto [a], Simon Thye Andersen [a], Richard Lin [b], Martin Schoeberl [a]

[a] *Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark*
[b] *Department of Electrical Engineering and Computer Sciences, UC Berkeley, Berkeley, CA, United States of America*
[c] *Department of Computer Science, ETH Zürich, Zürich, Switzerland*
[d] *Electrical Engineering Unit, Tampere University, Tampere, Finland*

ARTICLE INFO

ABSTRACT

With the current ever-increasing demand for performance, hardware developers find themselves turning ever-more towards the construction of application-specific accelerators to achieve higher performance and lower energy consumption. In order to meet the ever-shortening time constraints, both hardware development and verification tools need to be improved.

Chisel, as a hardware construction language, tackles this problem by speeding up the development of digital designs. However, the Chisel infrastructure lacks tools for verification. This paper improves the efficiency of verification in Chisel by proposing methods to support both formal and dynamic verification of digital designs in Scala. It builds on top of ChiselTest, the official testing framework for Chisel. Our work supports functional coverage, constrained random verification, bus functional models, and transaction-level modeling in a verification library named ChiselVerify, while the formal methods are directly integrated into Chisel3.

1. Introduction

General-purpose processors performance increase is down to single digit percentage per year. This leads to a new golden age for computer architects designing domain-specific hardware accelerators for future performance improvements [1]. The design of these accelerators is often complex, and their development is time-consuming and error-prone. To mitigate this issue, we shall learn from software development trends such as agile software development [2], and adapt to agile hardware development [3]. One move in this direction is Chisel [4,5], a Scala-embedded hardware construction language, that was introduced in order to move digital circuit description to a more software-like high-level language.

Hardware design is still dominated by the traditional hardware description languages, such as VHDL and SystemVerilog. SystemVerilog adds object oriented features to the language. However, those features are only available for test benches, not for describing hardware. Chisel goes one (or several) steps forward in hardware description, as it enables object-oriented and functional programming to describe hardware. This allows one to not only concisely describe hardware, but also to describe hardware generators. This elevates current practice of Perl, TCL, or Python generating VHDL or SystemVerilog code to use the powerful programming language Scala.

However, Chisel does not yet provide efficient verification tools or libraries. The ChiselTest package [6] only provides primitives to write classical test benches. Therefore, we build upon ChiselTest and add verification features. Those features are inspired by the Universal Verification Method (UVM), but implemented by leveraging Scala's conciseness and support for both object-oriented and functional programming. ChiselVerify, supports both coverage-oriented and constrained random verification flows with more features than those available in UVM. Additionally, formal methods for verifying chisel designs using bounded model checking are proposed enabling the formal verification of Chisel designs based solely on a specification. These two proposals combined make Chisel's verification capabilities on par with industry standards such as UVM.

For the evaluation, we use three designs: (1) the execution stage of the Leros processor [7], (2) an arbitration circuit, and (3) an industrial use case, a min-heap sorting circuit. We show that ChiselVerify can check many features of the min-heap with few lines of verification code.

The contributions of this paper are:

- Tools for defining and gathering functional coverage information about a Chisel design.
- A Domain Specific Language for constraint programming inside a ChiselTest test bench.
- A Bus Functional Model for the AXI4 standardized interface.[2]
- Formal methods[3] for verifying Chisel designs.

This paper is an extension of [8]. These extensions include the following:

- A proposal of formal verification methods for verifying Chisel designs.
- An additional functional coverage approach through implicit verification plans.
- An in-depth evaluation and comparison between our methods and those available in UVM and cocotb [9].

The paper is organized into 7 sections. First, Section 2 describes related work, and Section 3 covers background on hardware verification. Section 4 describes our solution for enabling verification in Chisel, called ChiselVerify. Section 5 presents an orthogonal approach to classic test driven verification by using formal methods for verification. Section 6 evaluates ChiselVerify with three use cases, and Section 7 concludes the paper.

2. Related work

Most new designs are being described in SystemVerilog; an extension of traditional Verilog which introduces many non-synthesizable elements. These extensions include various object-oriented programming constructs and are intended to allow for writing more advanced test benches. However, contrary to Chisel, the object-oriented design approach cannot be used for hardware description. SystemVerilog also offers constructs for gathering statement and functional coverage information [10], but our solution differs from these in several ways. In addition to SystemVerilog's range- or transition-based bins, ChiselVerify's `cover` constructs support temporal relations as well as generalized conditional bins based on user-defined predicates.

The temporal relation definition capabilities found in both ChiselVerify and its formal counterpart can also be seen in the Property Specification Language (PSL) and the similar SystemVerilog Assertions [11], which are two current solutions allowing for the use of temporal logic in relation to both coverage and assertions. Our approaches, however, differ in many ways, but are still all based around concepts taken from Linear-Temporal Logic (LTL) such as the `past` operator [12]. For example, PSL bases itself on a wide variety of Sequential-Extended Regular Expressions (SEREs), which define temporal relations between different boolean expressions. SEREs, however, are quite complex and require the use of many operators to describe potentially simple temporal relations. In contrast, our solution aims to provide a simplified set of temporal constructs to express LTL. These encompass a similar range of relations when used in conjunction with different types of bins in our coverage tools.

Cocotb [9] is a Python-based verification framework for VHDL and Verilog designs. It is enabled by extensions to Python for coroutine support and DUT interfacing using a very simple `dut.port.value` interface. Cocotb was also extended with a library called cocotb-coverage,

which allows for functional coverage and constrained random verification to be added to the Python-based test benches. This solution follows the same goals as ours, since it aims to improve verification efficiency by allowing for verification to take place in a high-level environment. However, our solution aims to do so in a way that is more closely integrated into the design flow, by allowing for a high-level design to be verified in an equally high-level environment. Cocotb, in contrast, makes verification happen in a completely separate environment as the design process. In a later section, we will compare ChiselVerify's functionalities to those available in cocotb.

Adding to the verification features introduced in SystemVerilog, designers also have access to the UVM, which was designed to be a standardized way of writing SystemVerilog test benches by focusing on both horizontal and vertical re-use [13]. Unfortunately, its generality leads to an inherent verbosity, i.e., even simple tests require at least around 800 lines of code. As a result, UVM imposes significant initial time-investment demands but is re-usable once it gets up and running. Moreover, newcomers may experience that UVM is less accessible than simpler approaches such as ChiselVerify as its structure differs from most traditional test benches.

Other works have focused on proposing different verification tools for other HCLs, such as bluecheck [14] for the BlueSpec HDL, or fault [15] which introduces verification components for the Magma HCL. Similarly to ChiselVerify, these solutions propose verification functionalities for their specific HCLs. Bluecheck focuses on enabling the creation of generic testbenches, similarly to UVM, that are synthesizable. Fault focuses on providing formal and constraint random verification functionalities, similarly to ChiselVerify, but embedded in Python.

Yosys [16] is an open-source tool for circuit synthesis. Claire Wolf developed the Yosys Open SYnthesis Suite (Yosys) as part of her Bachelor thesis at the Vienna University of Technology [17]. The focus of Yosys is on high-level digital synthesis. Yosys uses the open-source logic synthesis tool ABC [18] for gate-level optimizations. Yosys is a framework that can be extended. We can use Yosys for interactive design investigation, circuit analysis, or symbolic model checking.

Beyond the general frameworks available in SystemVerilog and UVM, other projects have proposed using software testing techniques to do hardware verification. For example, RFuzz [19] is a generalized method that enables efficient "coverage-guided fuzz mutational testing". It relies on FPGA-accelerated simulation and novel techniques for deterministic memory resetting to use fuzzing (i.e. randomized testing with dynamic seeding based on achieved coverage results) on digital circuits. Specifically, RFuzz automates collection of and adjustments based on branch coverage. In comparison to RFuzz, ChiselVerify offers a different type of solution focusing on implementing verification functionalities in a language while RFuzz offers an efficient way of using these to ameliorate testing; particularly by using coverage tools to guide its randomized verification. For the sake of completeness, we also mention another work of ours presenting the use of ChiselVerify's coverage tools for functional-coverage driven mutation-based fuzzing [20], as well as a similar work, which proposes a coverage-driven mutation-based fuzzer for SpinalHDL [21].

The SecChisel framework [22] adds security labels and static type checking of information flow to the Chisel languages. The type checks are performed using the Z3 SMT solver [23]. This is different from bounded model checking which verifies user defined assertions over signals and is able to find sound multi-cycle counter examples to failing properties. However, bounded model checking is incomplete, meaning we can miss bugs that only occur many cycles into the design execution. Static information flow tracking only looks at the combinatorial logic only to ensure that now information is leaked. This allows it guarantee that no information will be leaked, but can lead to false positives where the type check fails, but no bug exists.

To the best of our knowledge, ChiselVerify, along with the formal methods proposed in this work, forms the only framework that provides
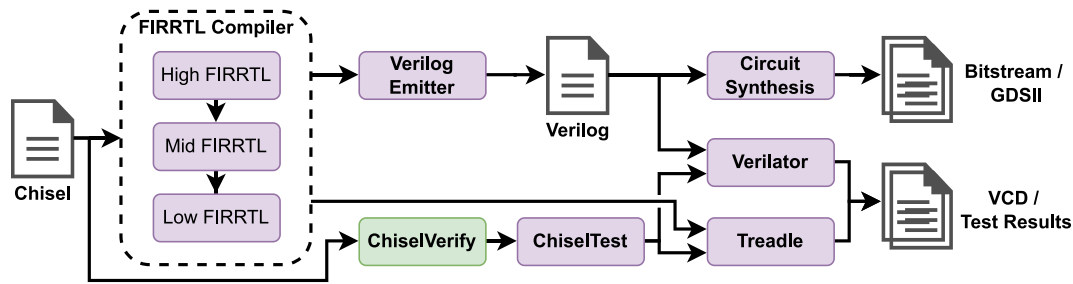

Fig. 1. Overview of the Chisel compilation pipeline.

easy-to-use formal and dynamic verification functionalities which are well-integrated into the Chisel and ChiselTest ecosystem.

The CHA library [24] implements temporal assertions similar to SystemVerilog Assertions for Chisel designs. It builds upon the formal verification framework which we discuss in Section 5.

3. Background

Before going into details of ChiselVerify, we provide a brief overview of hardware verification, and Chisel and its related existing verification techniques.

3.1. Verification of digital designs

Verification of digital designs refers to testing done before tape-out through simulation or (FPGA-based) emulation [10]. SystemVerilog is one of the main languages used for verification. It allows for engineers to define constraint-driven randomized test benches and metrics to gather functional coverage resulting for a suite of tests. We are interested in three verification features: functional coverage, constrained random verification, and bus functional modeling—all three of which are available in SystemVerilog, but rather complex to use as they are embedded in a low-level language.

3.1.1. Functional coverage

One of the most frequently used verification tools is test coverage which enables measuring progress and effectiveness of testing processes. In contrast to the common, quantitative statement coverage metric, which measures how many lines of code have been tested, functional coverage is qualitative and aims at answering which functionalities have been tested [10]. This enables measuring how correctly a design implements its specification, and it is measured relative to a verification plan which includes the following:

- `Bins` that declare ranges of values that should be tested for (i.e., expected values of a given port), and
- `cover` constructs specifying ports that need to be sampled in the coverage report, defined using a set of `bins`.

3.1.2. Constrained random verification

Using constrained random verification features, a verification engineer can create random variables constrained to specified sets of values. In doing so, even a relatively small test suite can, statistically, cover many functionalities of a design. Moreover, constraining inputs ensures that no unnecessary tests are run for input combinations that would not appear during regular operation [25].

A set of constrained random variables define a Constraint Satisfaction Problem (CSP). In CSPs, problem entities are represented as a finite, homogeneous collection of constraints. CSP solvers seek solutions to such problems and, thus, serve as the basis for constrained random verification.

Constrained random data types are native and declared with the `rand` keyword in SystemVerilog. Implementations of SystemVerilog simulators have a built-in CSP solver allowing for randomization through its `randomize` method.

3.1.3. Bus functional models

Abstraction is often the key to solve complex problems. Bus functional models represent such a technique. They implement models of (standardized) interfaces, like the Advanced eXtensible Interface version 4 (AXI4) from ARM [26], that enable interacting with either master or servant components at a transaction level that abstracts away bit-fiddling of individual wires. Digital hardware vendors often provide IP generators whose output blocks are equipped with such interfaces. Well-developed bus functional models enable simpler, safer, and less verbose interactions with such components.

3.2. The Chisel hardware construction language

Chisel is a "hardware construction language" embedded in the general purpose programming language Scala [5,27]. It allows designers to effectively write Scala programs that generate hardware descriptions at the Register Transfer Level (RTL). Compared to traditional hardware description languages, like VHDL and Verilog, Chisel is much more high-level and allows for object-oriented and functional programming in the context of digital design. One popular open-source application is the powerful RocketChip system on chip generator [28].

The user-facing API of Chisel is a Scala library with some syntactic sugar that allows the user to generate RTL designs. These designs then have to be converted into a format that is understood by simulators as well as FPGA and ASIC synthesis tools. The lowering is done by the FIRRTL compiler which converts a high-level Intermediate Representation (IR) into a normalized structural representation [29]. The low-level representation is then exported into a subset of Verilog that was chosen as a common subset supported by the majority of backend tools.

Besides serving as a convenient way to lower Chisel circuits into Verilog, the FIRRTL IR and accompanying compiler infrastructure also makes it easy to add circuit analysis and instrumentation passes, known as *transforms*. Moreover, it is possible to simulate circuits described in FIRRTL using the Treadle execution engine, before converting them into (System)Verilog which can, of course, be simulated using commercial or open-source tools like Verilator [30]. Fig. 1 shows an overview of the Chisel compilation pipeline.

Being embedded in Scala, Chisel is executed on the Java virtual machine (JVM) and can use existing Scala and Java libraries for design and verification. The JVM also allows for the use of the Java native interface for calling C functions, thus enabling co-simulation of Scala testers, Chisel designs, and a C-based golden models. This is valuable for companies wishing to keep their existing C models while using Scala/Chisel for simulation.

3.3. Testing Chisel designs

A Chisel design can be tested with ChiselTest [6], a non-synthesizable testing framework for Chisel that emphasizes simplicity while providing ways to scale up complexity. Like Chisel, ChiselTest is a Scala library that provides an interface into several simulators through `peek` (read value from circuit), `poke` (write value to circuit),

and `step` (advance time) operations. Tests written with ChiselTest are just imperative Scala programs that run one line after another.

However, ChiselTest is missing fundamental functionalities that improve verification efficiency. For example, it does not provide either of the three aforementioned features: functional coverage, constrained random variables, or bus functional models, despite these features being crucial for efficient verification.

4. Verification with chisel

As an extension of ChiselTest, we propose ChiselVerify, which introduces verification functionalities to the Chisel ecosystem. ChiselVerify bases itself on ChiselTest by using its design interfacing features in order to enable various verification functionalities directly in Chisel, such as functional coverage, constrained random verification, and bus functional modeling. In the following subsections, we present how we achieved our solution. We start by presenting its functional coverage capabilities, using both implicit and explicit specifications. We then propose constrained random verification tools for automating testing of general designs. Finally, we look at how bus functional models can increase testing ease, and demonstrate our method by creating a model of the standardized AXI4 interface.

4.1. Coverage in chisel

ChiselVerify allows for functional coverage constructs to be defined over a Chisel design directly in Scala. This is possible directly inside of a ChiselTest test bench, and enables the gathering of information about completeness of a test with relation to a given specification defined using a verification plan. In order to enable functional coverage in Chisel, we explore and define two methods for describing a specification, using an *implicit* or an *explicit* verification plan. These two methods are defined as follows:

- **Implicit verification plan:** Omit the verification plan, and obtain coverage data through queries on the ports we want to check *after* the test bench.
- **Explicit verification plan:** Declare the ports that will be sampled *before* the test bench.

Both methods have their advantages and disadvantages, which we will discuss in detail in a later section. In both cases, in order to define our methods, we needed to be able to do the following:

- Create a verification plan, either implicitly using a `Queryable-Coverage` object, which creates a verification plan containing all of our device's ports, or explicitly using `cover` constructs,
- sample our tested device's ports, using ChiselTest's interfacing capabilities,
- keep track of the different sampled values that hit for each bin, using a coverage database, and
- compile all of the results into a comprehensible and programmatically useful coverage report.

We will describe how the two different solutions are used to gather functional coverage information.

4.1.0.1 Implicit verification plans: As presented above, our solution enables the gathering of functional coverage information without explicitly having to define a specification. This is enabled through the use of a `QueryableCoverage` object, which is simply defined on a given DUT. The object must be defined within a ChiselTest test bench, allowing for the existence of a usable DUT. Once that is done, it must be sampled throughout the test suite using its `sample` method. Doing so will store the current value of every port in the DUT, thus enable for a future coverage queries. Obtaining coverage information using this method is done at the end of a test suite using coverage queries of the form of `get(port, expectedHits, range)`, where the arguments have the following meaning:

- `port` represents the DUT port for which we want coverage information.
- `expectedHits` is optional and represents a specification of number of hits we would expect for this port.
- `range` is also optional and represents the range in which we want to sample the port.

A query of this sort yields a `CoverageResult` case class that can then be used either programmatically or to generate a readable report using its `print` method. One can also simply print out a full report of all ports in the DUT using the `QueryableCoverage` object's `printAll` method.

```
1  val coverage = new QueryableCoverage(dut)
2  for (fun <- 0 until 50) {
3    dut.io.a.poke(toUInt(fun))
4    dut.io.b.poke(toUInt(fun % 4))
5    coverage.sample()
6  }
7  coverage.get(dut.io.outA, 50).print()
8  coverage.get(dut.io.outB).print()
9  coverage.get(dut.io.outA, range = 0 to 4).print()
10 coverage.printAll()
```

Listing 1: Example use of a `QueryableCoverage` object in order to gain information about the DUT's testing process. Note that `outA` and `outB` simply output the values of `a` and `b`.

Listing 1 shows a basic use of the implicit coverage tool. The above example outputs the following coverage Report:

```
Port io_outA has 50 hits = 100.0% coverage.
Port io_outB has 4 hits.
Port io_outA for Range 0 to 4 has 5 hits
= 100.0% coverage.


========= COVERAGE REPORT =========
Port io_outB has 4 hits.
Port io_outA has 50 hits.
Port io_b has 4 hits.
Port io_a has 50 hits.
===================================
```

This report shows the coverage results from our simple test in the form of the number of hits associated to each port. Given a specification for the amount of expected hits, the tool also shows a coverage percentage over the port. We can also see that given a range, the tool is also able to output a coverage percentage using the range length as the expected number of hits.

The `QueryableCoverage` object can be a useful tool for gathering simpler coverage information on a DUT. However, in order to define complex specifications for our functional coverage, explicit verification plans must be used.

4.1.0.2 Explicit verification plans: As detailed in our previous work on the topic [31], our solution allows for the definition of a target specification using a set of `cover` constructs defined inside of a `CoverageReporter`. This reporter functions as a front-end element that allows for the definition of complex specifications using only high-level constructs. The main interface is the `register` method, which stores `cover` construct to `bin` mappings inside of a `CoverageDB` database object, and groups them as a single `covergroup`. After the definition of the verification plan, the registered elements of the specification must be sampled explicitly by the user using the `sample` method, which checks the value of the associated port at that simulation cycle against the bins in order to determine if the specification is met for that value or not. Once the sampling has been done, at the end of a test suite, a functional coverage report is generated by the reporter using the `report` method, which interprets the results

stored in the database and compiles them to into a Scala `case class`. This report object can then be used to check for coverage thresholds, i.e., whether or not the coverage has surpassed a specific amount, or to guide the mutation of inputs in a mutation-based fuzzer, as was explored in previous work [20].

```
1 val cr = new CoverageReporter
2 cr.register(
3   cover("accu", dut.io.accu)(
4     bin("lo10", 0 to 9),
5     bin("First100", 0 to 99)),
6   cover("test", dut.io.test)(
7     bin("testLo10", 0 to 9)),
8   cover("accuAndTest", dut.io.accu, dut.io.test)(
9     cross("both1", 1 to 1, 1 to 1))
```

Listing 2: Small verification plan defined using 3 `cover` constructs, including one cross coverage construct

Listing 2 shows a basic verification plan defined using our functional coverage tool. This example shows the use of `cover` constructs with both a single and multiple ports. The use of multiple ports defines a *cross coverage* relation between the two ports, meaning that sampled values are considered a *hit* if they meet all specifications simultaneously [10]. Concretely, in our example, a hit would be considered if, within the same cycle, both `dut.io.accu` and `dut.io.test` are sampled with the value 1.

After defining our specification in the form of a verification plan, we need to define the ideal location, within our test bench, to sample the registered `cover` constructs using the reporter. The reporter's `sample` method can be used to sample either all groups simultaneously or simply one group at a time, by specifying the `id` of the group we want to sample in the method call. All `cover` constructs within a group will always be sampled together. Sampling different groups at different locations throughout the test suite can allow for more targeted coverage information to be gathered. Finally, once we are done sampling throughout the test suite, we can print out a readable coverage report by calling `cr.printReport()` which, for our example, results in the following:

```
=============== COVERAGE REPORT ===============
================= GROUP ID: 1 =================
COVER_POINT PORT NAME: accu
BIN lo10 COVERING 0 to 9 HAS 8 HIT(S) = 80%
BIN First100 COVERING 0 to 99 HAS 9 HIT(S) = 9%
===============================================
COVER_POINT PORT NAME: test
BIN testLo10 COVERING 0 to 9 HAS 8 HIT(S) = 80%
===============================================
CROSS_POINT accuAndTest FOR POINTS accu AND test
BIN both1 COVERING 1 to 1 CROSS 1 to 1 HAS
1 HIT(S) = 100%
===============================================
```

This report shows the three `cover` constructs that we registered and the associated hits that each of their `bins` has obtained. These hits represent the number of times that registered port was sampled with a unique value found within the given range. Each `bin` is then given a coverage percentage, based on the ratio between the number of hits obtained and the total number of possible values in the range.

In addition to the basic `cover` constructs presented until now, our tool allows for the definition of more complex relations, including delayed cross coverage, and purely conditional coverage. Inspired by concepts from Linear Temporal Logic [12], delayed cross coverage allows the user to define a relation between two ports sampled at different clock cycles. The idea is similar to how a `cross` works, but this time rather than sampling both points in the same clock cycle, we compare one port, at the starting clock cycle, to another port sampled a given number of clock cycles later. The temporal separation between the two ports is defined using a `delay`, for which we have defined four different types:

- `Exactly`, a hit is obtained if a value sampled from the second port meets its specification, as defined in the bin, exactly the given number of clock cycles after the first point was sampled.
- `Eventually`, a hit is obtained if the second port meets its specification at any point within the given number of clock cycles after the first point was sampled.
- `Always`, a hit is obtained if the second port meets its specification at every clock cycle for a given number of clock cycles after the first point was sampled.
- `Never`, a hit is obtained if the second port never has a sampled value that meets its specification during any clock cycle for a given number of clock cycles after the first point was sampled.

The use of timing-related bins is only possible if the coverage reporter is used to step the clock instead of ChiselTest's regular interface, since it allows for the `CoverageDB`'s internal clock to remain correct.

Conditional coverage enables the definition of bins containing fully custom hit-consideration rules using a user-defined predicate. Using this type of construct, one can check for arbitrary relations between an arbitrary number of ports. For example, it is possible to create a bin that checks for every clock cycle where all fields in a vector are equal. This is done by using a function of type `Seq[BigInt] => Boolean` in the `bin` declaration. The report then shows the number of distinct sampled values for which the predicate held throughout the testing process. Given the unbounded nature of these bins, an "expected number of hits" argument is required for each condition in order to yield a final percentage alongside the number of hits. Using these different types of `cover` constructs, one can express the specification of virtually any design.

4.1.0.3 Explicit vs. Implicit verification plans: Comparing the two solutions, Explicit verification plans enable more precise and complex definitions of specifications than implicit verification plans. This is due to the computational load inherently present in Implicit verification plans. Since Implicit verification plans require sampling over the entire range defined by the bit-width of every port, it is limited to use with simpler designs. However, implicit verification plans do not require the re-simulation of a design in order to obtain different coverage information, while explicit verification plans do. All in all, these coverage tools allow for the gathering of functional coverage directly in Chisel, thus filling in one of the holes inside of Chisel's ecosystem. Another tool our solution brings to Chisel, and enables one of the uses of functional coverage presented earlier in this section [20], is constrained random verification, which is detailed in the following paragraph.

4.2. Constrained random verification

A coverage-driven verification suite is not complete without access to randomization tools. ChiselVerify thus provides a set of tools, inspired by those available in SystemVerilog, which allow for the declaration and randomization of random objects. This is done by providing a "constraint programming" domain-specific language that runs on an existing CSP solver named JaCoP [32].

4.2.1. ChiselVerify's constraint programming DSL

Our solution allows for the declaration of a constrained random object by defining a `class` that extends the `RandObj` trait. The newly defined `RandObj` class contains all of the constraints and random variables that will later be used in our constrained random tests. This information will be stored inside of a `Model`, that is given to the `RandObj` on initialization. A `Model` is simply an object which functions as a database for our generated random number. It can be initialized with a seed to allow for predictable pseudo-random number generation. In total, there are two main elements that are used inside of the object to drive the randomization process: random variables and constraints.

4.2.1.1 Random variables: The first element represents a random value generator and is associated to constraints that will determine the set of values that the random variable can take. Our DSL allows for the declaration of two different types of random variables:

- *Regular,* can take any value that satisfies the associated constraints.
- *Cyclic,* cannot take the same values twice until the entire set of valid values has been explored.

Similar to the interface designed for the functional coverage tools, both types are declared using a single unified function call `rand` to which we give a lower and an upper bound on the values the variable can take. As an example, `rand(0, 5, Cyclic)` will declare a random variable that will yield six distinct values in a row before starting to repeat itself.

4.2.1.2 Constraints: Our solution allows for the definition of both single constraints and `ConstraintGroups`. Constraints are defined by applying constraint operators on random variables. Additionally, the `IfCond` and `ElseC` constructs allow for the definition of conditional constraints. Every defined constraint may be enabled or disabled at any point in the test suite. This can also be done to multiple constraints simultaneously by enabling or disabling a `ConstraintGroup`.

4.2.1.3 Using a RandObj: After declaring and filling random objects with random variables and constraints, a `RandObj` must be instantiated and then randomized using the `randomize` method within a test bench. The `randomize` method's return value is predicated on the solvability of the constraints by the CSP solver. Once randomized the random variables within a `RandObj` yield a valid random value when prompted with their respective `value()` methods.

```
1  class Packet extends RandObj(new Model(3)) {
2    val idx = rand(0, 10)
3    val size = rand(1, 100)
4    val len = rand(1, 100)
5    val payload: Array[Rand] = Array.tabulate(11)(
6      rand(1, 100)
7    )

9    // Example Constraint with operations
10   val single: Constraint = (payload(0) == (len - size))

12   // Example conditional constraint
13   val conditional = IfCon(len == 1) {
14       payload.size == 3
15   } ElseC {
16       payload.size == 10
17   }
18   val idxConst = idx < payload.size
19 }
```

Listing 3: Example usage of a random object. `rand(min, max, type=Normal)` declares a random variable. Any operation on a random variable generates a constraint.

Listing 3 shows an example use of the different ways one can define a random variable with constraints. As seen in the example, collections of random variables, such as arrays, can be defined and constraints can be placed on the collections themselves. This approach is seen in the `payload` random variable, where a constraint is placed on the size of an array of random variables. The `conditional` random variable shows how conditional constraints can be declared. In our example, the constraint placed on `payload` depends on the value of the `len` random variable.

These constrained random objects are a powerful tool that can be combined with the aforementioned coverage functionalities to create coverage-driven randomized tests. With the use of our solutions, such a setup greatly improves the automation capabilities of Chisel test

benches. However, these capabilities may be further improved by abstracting away groups of wires and operating on an operation or transaction level instead.

4.3. Verification with bus functional models

Component re-use, portability and flexibility are also interesting in the context of digital designs. A good way to achieve these characteristics is by equipping one's designs with standardized interfaces. Verification engineers can test such components at a transaction level by combining constrained random verification and coverage measures with bus functional models. As an example, we provide a bus functional model for AXI4, an open standard by ARM [26].

4.3.1. Introduction to AXI4

The AXI4 protocol by ARM is a generic, flexible interconnect standard. It comprises five independent handshake-based channels; three for write operations (*Write Address*, *Write Data*, and *Write Response*) and two for read operations (*Read Address* and *Read Data*). Interconnect operations, known as transactions, consist of sequences of transfers across either set of channels. All channels share a common clock and reset.

As an example, consider a write transaction of 16 data elements. First, the manager provides the transaction attributes (e.g., target address and data size) as a transfer on the *Write Address* channel. Next, it transfers the data elements one at a time over the *Write Data* channel. Finally, the subordinate indicates transaction status on the *Write Response* channel. Note that channel independence means that data may be transferred before the transaction attributes and that the read channels may operate at the same time [26].

4.3.2. Implementation

To enable easy verification of AXI4-interfaced components using ChiselVerify, we provide definitions of channel wire bundles, abstract manager and subordinate classes, and a transaction-based bus functional model: the `FunctionalManager` class. Through parameterizing the manager with a `Subordinate` DUT provides a simple interface to control the DUT. Its two most important methods are `createWriteTrx` and `createReadTrx` to create and enqueue write and read transactions, respectively.

Our bus functional model implementation utilizes ChiselTest's multithreading features to allow for non-blocking calls to the aforementioned methods and for emulating the channel independence more closely. When a write transaction is enqueued and no other write transactions are in flight, the bus functional model spawns three new threads, one for each channel. The threads handle the bit-fiddling required to operate the channels.

4.3.3. A test example

Returning to the example of transferring 16 data elements, consider the test for a module called `Memory` listed below. First, a write transaction with 16 data elements takes just one call to `createWriteTrx` most of whose arguments have default values. It is equally simple to create a subsequent read transaction. Depending on the DUT implementation and due to channel independence, not waiting for a write to complete before initiating a read may return incorrect results.

```
1  class MemorySpec extends AnyFlatSpec with
       ChiselScalatestTester {
2    behavior of "My memory module"
3    it should "write and read" in {
4      test(new Memory()) { dut =>
5        val bfm = new FunctionalManager(dut)
6        bfm.createWriteTrx(0, Seq.fill(16){0x7FFFFFFF},
7          len = 15, size = 2)
8        bfm.createReadTrx(0, len = 15, size = 2)
9      }
```

```
10  }
11 }
```

Listing 4: Using the AXI4 bus functional model with ChiselTest

4.4. Future work

In this paper our development stays mainly in the Chisel world and comparing it against UVM. However, an interesting approach would be to mix and mach different approaches and tools. The easiest combination is to use Chisel for the circuit description and UVM for verification. Chisel generates standard Verilog code that can be integrated with an UVM testing flow. SystemVerilog components can also be integrated in Chisel as black boxes and we can verify them with ChiselVerify on Verilator. However, the SystemVerilog code shall only include Verilog code constructs that Verilator supports.

The most challenging approach would be to combine UVM and ChiselVerify. For example, having an UVM test driver for the DTU, but using ChiselVerify with a golden reference model written in Scala. We consider this interesting development question as future work.

5. Formal verification

An alternative to exercising the circuit description with a set of concrete inputs is to symbolically explore the circuit execution for any inputs for a limited number of cycles. This technique is called bounded model checking [33] and works by unrolling the circuit for k cycles and then asking a Boolean satisfiability [34] or SMT [35] solver whether there exists a set of inputs and starting states for the memories and registers in the design, for which an assertion is violated. If the solver returns a satisfying assignment to this query, we obtain a counter example that can be expressed as a test bench that initialized the state to concrete values from the solver and then drives the inputs for k cycles with the inputs obtained from the solver. If the solver returns that there is no such assignment, we get a guarantee that our circuit will not hit any assertion violation for the first k cycles of its execution.

There has been a long tradition of open-source formal verification systems from the academic community [36–38]. However, because of the traditional academic incentive structure, these research systems were hard to use or did not support enough features of the RTL design language to be widely used by a community of open source RTL designers. This changed with the introduction of the yosys [16] 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 [39] or aiger [40] formats without having to worry about supporting the much more complicated Verilog standard. The open-source SymbiYosys [41] tool wraps yosys as well as various formal verification engines in order to allow users to verify their designs. All a user has to provide are the Verilog sources of their design including assertions and assumptions as well as a small configuration script. SymbiYosys translates any failing traces it discovers into Verilog test benches and VCD waveform dumps for the user to inspect.[4]

In this paper we describe our approach to providing Chisel users with an easy way to formally verify their designs. We adapt many good ideas from yosys and build several new convenience features on top of them, taking advantage of the existing compiler infrastructure for Chisel. We first present two introductory examples and then provide details on how the formal backend was engineered to ensure that all failures could be replayed in simulation to help debugging in Section 5.3. We then discuss reset assumptions (Section 5.4) and a simple approach to temporal assertions using an improved version of the past statement which avoids common pitfalls compared to the equivalent functionality in Verilog (Section 5.5).
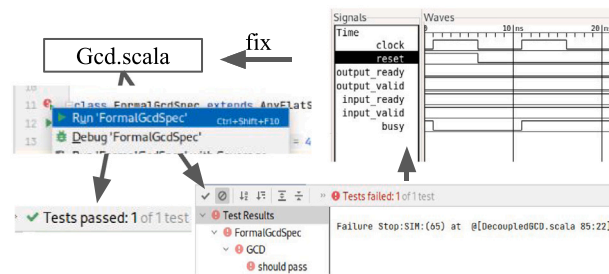

Fig. 2. Formal checks can be launched from a Scala IDE.

5.1. Example: Verifying a GCD circuit

We focus on a simple greatest common denominator (GCD) circuit which is a standard Chisel example.[5] Dynamic tests written with ChiselTest can be executed through a Scala IDE or from a shell with the `sbt test` command. With our proposed extension, formal checks can be run in a similar fashion. The user just needs to substitute the `test(new DecoupledGcd(16))` command with `verify(new DecoupledGcd(16), BoundedCheck(10))`. If the user now 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 additional program required is a copy of the open-source SMT solver Z3 [23].

Initially the check will always pass, no matter which changes we make to our circuit. Since the GCD circuit contains no assertions, there is nothing to tell the solver if the circuits misbehaves. In order to actually verify something, an assertion needs to be added 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 will pass because the circuit does indeed fulfill the property after reset.

We now introduce a small bug by connecting `input.ready` to `true.B` and rerun the test. An assertion violation will be reported one cycle after reset. The user is also presented 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 the ChiselTest library. Since we replay the test on a concrete simulator, the error message and VCD will be exactly the same as if the user was running a simulation test.

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 make use of our `past` function for temporal properties which is described in detail in Section 5.5.

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 the same way as any other unit test. A VCD waveform dump is automatically generated to help debug failing checks. This is illustrated in Fig. 2.

---

[4] With the open-source GHDL plugin, yosys now also supports formally verifying VHDL circuits.

[5] The GCD code is part of the Chisel template: https://github.com/freechipsproject/chisel-template.

cycle #0: read and write
issued from/to address 0

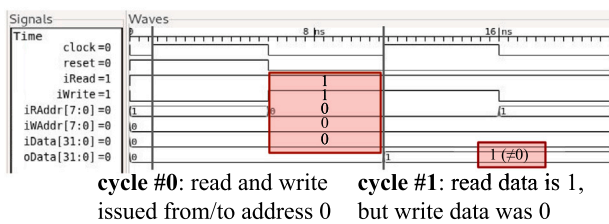**cycle #1**: read data is 1,
but write data was 0

Fig. 3. RTL code with inline assertion, formal test and counter example waveform for a verification example.

5.2. Example: Verifying the read behavior of a RTL memory

The example shown in Fig. 3 verifies that when a Chisel memory with synchronous read port and `WriteFirst` behavior has a read and a write access to the same address, the new value will be returned. The check fails if `WriteFirst` is substituted with `ReadFirst` or `Undefined`. It is based on a Verilog example from a popular blog.[6] In the Chisel version, the assertion is automatically delayed until at least one cycle after reset, when there are valid `past` values available. A bounded model check is executed by the `verify` command, which is called from a standard Scala unit test. When the check fails, the failing inputs and starting states are replayed on a simulator, resulting in a waveform file that is 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, two cycles after reset are needed to fail the property. The first cycle contains the read and write requests and the second cycle observes the arbitrary result on the read port if we set the memory behavior to `Undefined` for read/write conflicts. The included screenshot was obtained with the open-source GTKWave waveform viewer.

5.3. A formal backend for FIRRTL

In order to implement the verify command, introduced in the previous sections, we need to convert the Chisel circuit into a format that is understood by open-source model checkers or SMT solvers. We can do this by using the FIRRTL compiler to convert the circuit to Verilog and then using yosys to convert to the model checking formats. While we initially used this approach, we eventually decided that it would be better to add a formal backend to the FIRRTL compiler directly. This way we can avoid the complicated Verilog semantics, model circuit behavior in greater detail and easily replay counter example traces on our FIRRTL simulator called treadle.

Users want their Chisel designs to be implemented with as little hardware as possible. In order to allow for efficient implementations, the FIRRTL specification was crafted to allow some operations to result in arbitrary results. 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 that another port is writing to or writing from two memory ports to the same address all generate an arbitrary value result. Not all of these behaviors are represented in the generated Verilog. The compiler is free to substitute arbitrary values with (more) concrete values, like always returning a memory read result even when the read port is disabled or by assigning a priority to write operations so that at least one of them will complete. Thus if we first generate Verilog and then use yosys, we are only verifying one concrete translation of the design, but there may be other legal translations that would violate the property. This is relevant, 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 in order to generate better hardware. This is the reason why we decided to carefully model arbitrary values as part of the FIRRTL compiler's new formal backend.

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 large resources of a major EDA vendor, we would like to reuse as much of the existing simulator infrastructure as we can. If we can replay the failing trace on our existing simulator, the VCD waveform dump and the error reporting will be of the same quality as when writing a concrete test bench. In order to be able to replay failures caused by arbitrary values, we carefully engineered two FIRRTL passes that analyze the circuit and add wires to detect when a result is arbitrary as well as a mux to substitute the result with a connection to a `DefRandom` node in that case. The new `DefRandom` construct provides a named arbitrary value that can change every clock cycle, very much like a `anyseq` annotated wire in Verilog. The formal backend implements `DefRandom` nodes as inputs that can be freely chosen by the formal engine. To make `DefRandom` work with our simulator, we replace the nodes with registers of the same type that are never updated by the hardware. Instead we use the software interface to our simulator to update these registers with the values chosen by the formal engine in each cycle. Fig. 4 shows our compilation flow in more detail. 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.

The btor2 format does not support hierarchical circuits and we thus always flatten the system by inlining everything into a single module. In order to ensure that we produce a good waveform dump, the counter example will be replayed on the non-inlined circuit. We make use of the built-in annotation support of the FIRRTL compiler to automatically track 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.

Once the circuit has been flattened, the conversion to a transition system is fairly straight forward. We implemented a SMTLib and btor2 encoding that is 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 [42]. Our backend supports memory and register initialization using the same user annotations as the Verilog backend. Multi-clock support through a clock stuttering pass is work in progress, for now only circuits with a single clock domain are officially supported.

5.4. Reset assumptions

In Chisel, users rarely need to worry about resets. Registers with reset values are automatically connected to the default reset and module instances just inherit their reset domain from their parent. In Verilog, users have to manually ensure that assertions are only triggered after the circuit was properly reset. We decided to provide sensible defaults instead. Assertion statements are automatically disabled, just like it has been the case for print and stop statements since the early days of Chisel. As part of our formal verification support, we provide a FIRRTL pass that automatically adds a constraint for the reset of the top level module to be active during the first cycle of execution. Thus, by default, users do not have to worry about reset. Their assumptions will only fire after their circuit has been properly reset and hence we ensure that there are no false positives. We do 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.
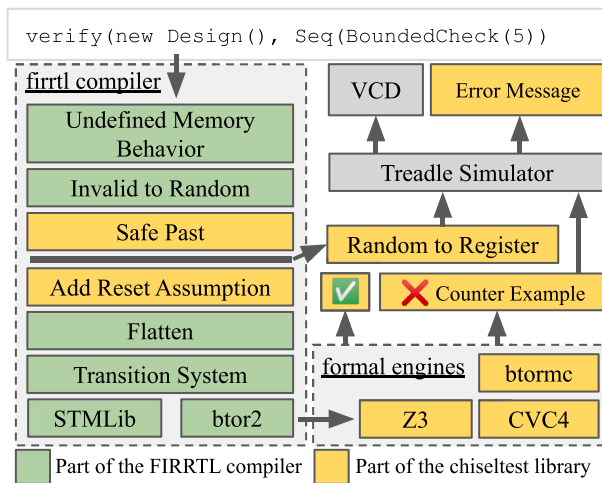
---

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

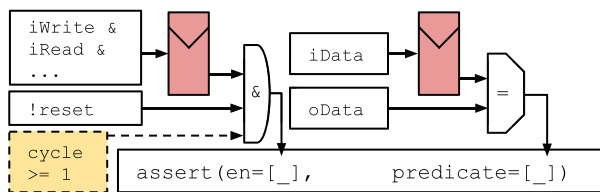Fig. 4. Our formal verification flow.

Fig. 5. Hardware generated by our tool to implement the temporal assertion in Fig. 3.

5.5. 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 are temporal assertion languages like SystemVerilog Assertions [43]. However, these are complex to implement efficiently and as of now there has not been a successful open-source implementation. 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.

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 frontend, 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 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 [44].

Fig. 5 shows how the temporal assertion from Fig. 3 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 past since the last reset. The new enable condition is derived from a synthesizable, saturating `cycle` counter which is created by the compiler pass.

6. Evaluation

In order to evaluate the different verification tools provided by ChiselVerify, we will be using three different example circuits, namely an accumulator ALU from the Leros processor [7], an arbiter circuit, and a priority queue provided by Microchip [45]. These are tested using ChiselTest test benches augmented with our verification tools. The test benches are then compared to similar tests written using SystemVerilog with UVM, and `cocotb`, using the generated Verilog descriptions as a DUT. We have selected UVM since it is an industry standard that can be used to verify Verilog designs, and `cocotb` since it enables similar verification functionalities in a high-level environment like what we propose with ChiselVerify. The results are then compared in terms of the verbosity of the verification code. As a final evaluation, we look at the overhead induced by the use of ChiselVerify-specific functionalities, such as functional coverage and constrained random verification, by comparing the runtime of a bare ChiselTest test bench, to the same one enhanced with our verification tools.

6.1. Evaluation circuits

We evaluate ChiselVerify on three distinct circuits, which we describe in the following paragraphs.

6.1.1. Leros accumulator ALU

The first circuit we will verify with our solution is an accumulator ALU from the Leros processor. It supports operations such as `add`, `load`, `shiftRight` and logic operations. In order to have a complete test suite, we need to try all available operations and use all potential input value corner cases. We will thus model our test bench to do so.

6.1.2. An arbitration circuit

Another one of the circuits we evaluate ChiselVerify on is an arbiter circuit. The arbiter uses a ready/valid interface for the clients and the shared resource. The arbiter is built as a binary tree, where each node does a local arbitration and contains a register to store the data until it can be communicated further up towards the root of the tree. Each local 2 to 1 arbiter has a turn flag to be fair between the two requests. The assumption is that this local fairness translates to a global fairness of the full arbitration tree.

To test the arbiter, we specify properties that result in a verification plan:

- Each request shall eventually be seen at the output (the root node).
- No requests shall be "generated" in the tree (out of thin air).
- The maximum latency for a request shall be n cycles without any competing requests.
- The maximum latency for a request shall be m cycles under full load.

- The arbitration shall be fair, which means the bandwidth difference between clients shall be bound by x %.

The values of n, m, and x, depend on the number of clients, i.e., the size of the arbitration circuit.

We have shown that the original implementation of `reduceTree` on the Chisel `Vec` produces an unfair circuit. Therefore, we have improved that implementation in Chisel. The pull request[7] is now available with Chisel Version 3.5.3.

6.1.3. Sorting in hardware

The final application example builds on a use-case provided by Microchip [45]. The use-case, which we implemented and verified, is a scheduling module for real-time systems built around a hardware priority queue. A host system can insert deadlines in the form of timestamps into the scheduling module which in turn presents the next upcoming deadline.

The scheduling module is implemented as a state machine operating on a set of memory banks. The memory contains a k-min-heap which is used by the internal priority queue to efficiently determine the next upcoming deadline. The implementation exploits parallelism to improve performance when fetching nodes from the heap memory by splitting siblings over separate memory banks. Furthermore, the search for the minimum key between a parent node and its children, a key operation when balancing the heap, can be parallelized by employing a reduction tree.

The presented constrained random verification and functional coverage functionalities of the ChiselVerify framework were used to verify the scheduling system and its submodules. The circuit at a top level allows for the insertion of deadlines into the scheduling system and the removal of them. As such the interface is not very complex and only consists of three flow-control pins as well as data pins used to provide a new deadline and to refer to a deadline which should be removed.

The test benches used to verify the scheduling system make use of random stimuli for the data pins and directed stimuli for the flow-control. We use the functional coverage report to check how well the stimuli driven on the DUT's data pins are spread over the spectrum of possible values, and to check whether certain input combinations are applied to the DUT at some point throughout the test. As an example, the timed coverage feature made it easy to check whether the `valid` input of the DUT was revoked at some point within 10 clock cycles after issuing an operation by adding the following cross-coverage group:

```
1 cover("timed_valid", dut.io.valid, dut.io.valid)(
2   Eventually(10))(
3     cross("revoked_valid_under_op", 1 to 1, 0 to 0))
```

Listing 5: A timed cover construct.

In order to check whether the DUT matches its specification, we have implemented a reference model for each module. At the top level, this model is written at a transaction level, while some submodules are tested against cycle accurate reference models. In order to abstract interaction with the DUT and provide an interface at the same abstraction level as the reference model, we employ bus functional model-like wrappers.

6.2. Evaluation results

We can now take a look at the different results obtained during our evaluation.

6.2.1. Verification verbosity

We start by comparing the three verification languages in terms verbosity, which is measured in "verification lines per source lines of code", a metric used in other works to partially evaluate verification languages [31,46,47]. We consider a verification line to be any explicit declaration of a cover or bin construct, as well as any function call or standard statement, and have formatted our listings accordingly. For this, we measure the overhead of the functional coverage and constrained random verification tools provided in ChiselVerify against those provided in UVM and `cocotb`.

We start by adding functional coverage to the simple test bench presented for the arbiter circuit. In order to do so, we need to define a verification plan that correctly captures information about the expected behavior of our DUT. The arbiter circuit takes as input a Vector of n `DecoupledIO` elements, which expands to $3n$ ports in the generated verilog. In order to conduct a coherent experiment, we set $n = 7$, and will define our specification accordingly. Our verification plan will thus cover the inputs and output `ready`, `valid`, and `bits` (data) ports.

We now do the same for the priority queue. As described above, this design has two interfaces we will want to monitor during our testing, i.e. the `cmd` and `head` interfaces. Unlike the arbiter, these interfaces are of fixed size and thus the chisel design and the generated Verilog will have similarly abstracted ports. Our verification plan will thus cover all elements of both the `cmd` and `head` interfaces.

6.2.1.1 Functional coverage in UVM: In order to gather functional coverage information about our design using UVM, we have to work with the generated Verilog description. Additionally, UVM requires a very specific test bench structure than spans multiple files and hundreds of LOC. Our focus is on the `uvm_subscriber` subclass, which is where our verification plan will be defined. For example, this is done for the arbiter circuit using the following structure:

- Create a UVM-subscriber based coverage class.
- Instantiate the current DUT (`Arbiter dut = new;`)
- Declare the verification plan, where a single port is covered with:

```
1   covergroup cg_input0;
2   IN0_READY: coverpoint dut.io_in_0_ready {
3     bins zero = {0};
4     bins one = {1};
5   }
6   IN0_VALID: coverpoint dut.io_in_0_valid {
7     //Same bins as in IN0_READY
8   }
9   IN0_BITS: coverpoint dut.io_in_0_bits;
10  endgroup cg_input0
11  // [...] Repeat for each port
```

- Define `build_phase` and `write` functions.
- Define the coverage class constructor.

Only considering coverage-related code, this requires 158 lines of SystemVerilog code.

6.2.1.2 Functional coverage in cocotb: Gathering functional coverage using cocotb is also done using the generated Verilog description. This is done by creating a `Python`-based test bench, which is linked to our design via a custom `Makefile`. We then define our verification plan by creating a `coverage_section` inside of our test bench. Finally, we link said verification plan to a `cocotb.coroutine` function to mark it as our sampler. For example, the arbiter circuit's verification plan has the following structure:

```
1  range_relation = lambda val_, bin_ : bin_[0] <= val_ <=
      bin_[1]
2 Arbiter_Coverage = coverage_section(
3    CoverPoint("top.io_out_ready",
        vname="io_out_ready", rel = range_relation,
```

CV vs. SV vs. PY Verification LOC

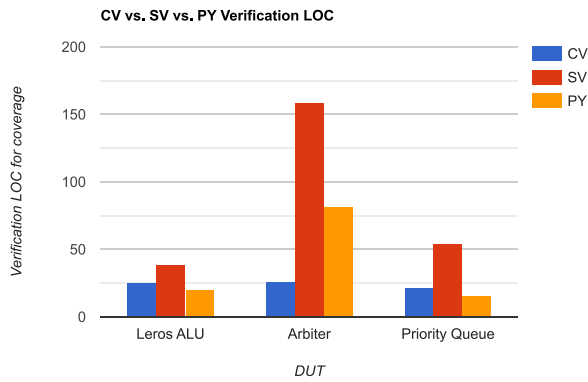Performance Overhead of ChiselVerify

Fig. 6. Verbosity overhead comparison between verification plans written for three designs with ChiselVerify (CV), SystemVerilog (SV), and cocotb (PY). For SV, only the `uvm_subscriber` code is taken into account.

Fig. 7. Performance overhead of the different tools provided by ChiselVerify. N represents the number of arbiter ports which also means the number of covergroups or random variables. The overhead is computed as the ratio between the average runtimes, over many runs, of the bare ChiselTest test bench and the same test bench augmented with either functional coverage (FC) or constrained random verification (CRV) for a given n.

```
4         bins = [0, 1], bins_labels = ["ready0",
            "ready1"]),
5     CoverPoint("top.io_out_valid",
            vname="top.io_out_valid", rel =
            range_relation,
6         bins = [0, 1], bins_labels = ["valid0",
            "valid1"]),
7     # [...] Repeat for each port
8 )
```

Here the range relation lambda tells cocotb how to interpret the bounds of a range. This can be compared to how ChiselVerify allows for custom hit conditions to be defined. When a single number is give, e.g. 0, this is interpreted as a range 0 to 0, thus a hit is considered if our relation holds for the sampled value, i.e. $0 <= val <= 0$. Only considering coverage-related code, the verification plan defined in cocotb requires 82 lines of cocotb-coverage code.

6.2.1.3 Functional coverage with ChiselVerify: Finally, using ChiselVerify to gather functional coverage information about a design allows for direct interaction with the Chisel design. In the case of designs with complex interfaces, such as the arbiter, this allows us to create generic verification plans that function with any parametrization of the design. For example, in the case of the arbiter, which has a variable number of inputs, we can simply define our verification plan using a generic loop over all inputs in the design as follows:

```
1 dut.io.in.zipWithIndex.foreach((input:
      DecoupledIO[UInt], idx: Int) => {
2   cr.register(
3     cover(s"in${idx}.ready", input.ready)(
4        bin("Ready0", 0 to 0),
5        bin("Ready1", 1 to 1)),
6     // [...] Continue the VP of a single input
7   )
8 })
```

Here, the `zipWithIndex` call allows us to concisely loop over our array with a `foreach` loop, while having access to the loop index, enabling the creation of unique identifiers for our `cover` objects. Only considering coverage-related code, this definition requires 26 lines of code for the arbiter's complete verification plan.

6.2.1.4 Summary: The results from our comparison are summarized in Fig. 6. It is important to keep in mind that these user-level comparisons are not absolute, and are used to give the reader an idea of the amount of code that is required to be written in the different languages to solve the same problems. Fig. 6 shows that, with ChiselVerify, we obtain on average a 70% reduction in LOC over UVM and a 38% reduction over cocotb. This is due to ChiselVerify's capability of exploiting the
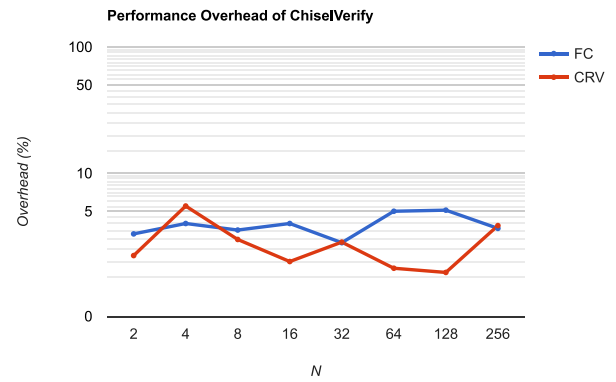
high-level design constructs, such as vectors, directly in the verification plan, while the other solutions need to rely on the unfolded generated Verilog. For designs with simple interfaces such as the priority queue or the Leros ALU, we obtain an increase of 30% over cocotb. This comes from cocotb's grouped bin declarations in a coverpoint. While it can be more concise to declare coverpoints in cocotb with simple designs, complex relations, often found in realistic designs, are more difficult to model. It is more direct to access the ports from a Chisel description using ChiselTest instead of UVM or cocotb, since we can use the Chisel object as it was originally described, and not have to interface via a more low-level generated Verilog description. This becomes especially important for deeper nested structures of IO ports.

6.2.2. ChiselVerify performance overhead

In order to evaluate our proposed solutions more thoroughly, we will look at the performance overhead that comes with using our verification tools to augment a ChiselTest test bench. To do so, we use our baseline a bare ChiselTest test bench without any functional coverage or constrained random verification.

6.2.2.1 Functional coverage performance overhead: In order to analyze the performance overhead over varying verification plan sizes, we will be using our previously mentioned arbiter circuit, which has a variable number of input ports. We start by measuring the performance of our arbiter test bench with n ranging from 2 to 256. We then measure the performance of the same test bench, but this time augmented with the verification plan presented earlier. Finally, we measure the performance of our initial test bench, but where the inputs of the arbiter are set using random variables from the following random object:

```
1  class ArbiterIn(n: Int) extends RandObj {
2    currentModel = new Model(seed)
3    val dins = for(_ <- 0 until n) yield(rand(0, n))
4    val consts = for(din <- dins) yield {
5      din dist (
6        0 to 0xF := 1,
7        0xF to 0xFF := 1,
8      )
9    }
10   // Enable all constraints
11   consts.foreach(_.enable())
12 }
```

The results from these measurements are summarized in Fig. 7. These results show that the both the functional coverage and constrained random verification tools scale quite well, since their overheads seem to be almost independent of the size of the verification

Table 1
Various details about the processor used for our benchmarks.

	Processor
Model	Intel core m5-6Y54 CPU
μArch	Skylake
Base frequency	1.20 GHz
Max frequency	2.7 GHz
N_Threads	4

Table 2
Runtime results obtained when running the `random_test` benchmark implemented with UVM and with ChiselVerify.

Runtime [s]	Mean	Std-deviation	Startup time
UVM	1.2	0.16	82.5
ChiselVerify	11.3	1.2	13.23

plan or the number of random variables used. They both show to have an average overhead of 3.5% over ChiselTest independent of the scale at which they are used. Such a low and scalable overhead shows that ChiselVerify is an efficient addition to the Chisel verification ecosystem.

6.2.3. ChiselVerify and UVM performance comparison

Finally, we evaluate our solution's performance by directly comparing the runtime of two tests, one implemented using SystemVerilog with UVM and the other with ChiselVerify. The UVM testbench is running using Vivado,[8] as Verilator has yet to support the full verification functionalities required by UVM.

6.2.3.1 Benchmarking infrastructure: To evaluate the performance of our solution, we run one of the tests of the Leros accumulator ALU presented in a previous section. The relevant test, `random_test`, stimulates the device under test using uniformly random inputs. This test is run 50 times in order to obtain a non-biased runtime measured in seconds. A description of the system used for our benchmarking can be found in Table 1.

6.2.3.2 Benchmarking results: The results from running the test using first UVM, and secondly using the various tools provided in ChiselVerify, are compiled and summarized in Table 2.

These numbers show that ChiselVerify's reliance on ChiselTest and sbt gives it a rather slow runtime when testing, but with a much faster startup time, making it ideal for smaller test suites. Recent work by Iyer et al. [48] has shown that this might be due to the way ChiselTest handles multi-threading, and efforts, also presented in part in said work, are being made to alleviate this overhead.

6.2.4. Evaluation summary

As demonstrated above, ChiselVerify allows for the use of functional coverage constructs, as well constrained random verification directly inside of a ChiselTest test bench. Complex constructs can be defined following the low verbosity of Chisel and ChiselTest, requiring on average a 70% fewer LOC than the industry standard UVM and a 38% fewer than cocotb. ChiselVerify improves upon these existing solutions by enabling direct use of high-level aspects of a Chisel design. All of this is done with a low performance overhead, averaging at 3.5% across the different functionalities. Iyer et al. [48] have shown that ChiselTest's command api performs better than cocotb's by a factor of 2x. However we have seen that, given its low startup time, ChiselVerify still remains slower, in terms of post-startup runtime, than UVM. With the efforts presented in Iyer et al.'s work, we can expect the performance of ChiselTest, and thus also ChiselVerify, to improve in the near future.

---

[8] https://support.xilinx.com/s/article/1070861.

7. Conclusion

In this paper, we proposed well-integrated tools for verifying Chisel designs directly inside of a ChiselTest test bench. These include functional coverage and constrained random verification tools, as well as bus functional models all under a single library named ChiselVerify. We also proposed formal verification methods, which are directly integrated into Chisel3, allowing for bounded model checking to be done in Chisel. Using these tools on three different designs of varying complexity, we obtained similar results to those of using UVM or cocotb, all while requiring less verification code. We also showed that the additional use of our tools within a ChiselTest test bench induces very little performance overhead. With this, we enabled functional coverage, constrained random verification, bus functional models and formal verification techniques to be used in the Chisel/Scala ecosystem. This will hopefully improve current verification engineering efficiency all while easing the way for software engineers to join the hardware verification world.

Source access

This work is in open source and hosted at GitHub: https://github.com/chiselverify/chiselverify. We plan also to regularly publish it on Maven.[9] The formal verification methods proposed in this work are integrated into Chisel3: https://github.com/chipsalliance/chisel3

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data/code for this work is open source and freely available.

References

[1] W.J. Dally, Y. Turakhia, S. Han, Domain-specific hardware accelerators, Commun. ACM 63 (7) (2020) 48–57.
[2] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R.C. Martin, S. Mellor, K. Schwaber, J. Sutherland, D. Thomas, Manifesto for Agile software development, 2001, https://agilemanifesto.org/.
[3] J.L. Hennessy, D.A. Patterson, A new golden age for computer architecture, Commun. ACM 62 (2) (2019) 48–60.
[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, K. Asanovic, Chisel: constructing hardware in a Scala embedded language, in: The 49th Annual Design Automation Conference (DAC 2012), ACM, San Francisco, CA, USA, 2012, pp. 1216–1225.

---

[9] https://mvnrepository.com/artifact/io.github.chiselverify/chiselverify.

[5] M. Schoeberl, Digital Design with Chisel, Kindle Direct Publishing, 2019, available at https://github.com/schoeberl/chisel-book.

[6] R. Lin, K. Laeufer, Chiseltest, 2022, https://github.com/ucb-bar/chiseltest.

[7] M. Schoeberl, M. Petersen, Leros: The return of the accumulator machine, in: M. Schoeberl, T. Pionteck, S. Uhrig, J. Brehm, C. Hochberger (Eds.), Architecture of Computing Systems - ARCS 2019 - 32nd International Conference, Proceedings, Springer, 2019, pp. 115–127.

[8] A. Dobis, T. Petersen, H.J. Damsgaard, K.J.H. Rasmussen, E. Tolotto, S.T. Andersen, R. Lin, M. Schoeberl, ChiselVerify: An open-source hardware verification library for Chisel and Scala, in: 2021 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), 2021.

[9] A.M. Smith, J. Mayo, R.C. Armstrong, R. Schiek, P.E. Sholander, T. Mei, Digital/analog cosimulation using CocoTB and xyce, 2018, http://dx.doi.org/10.2172/1488489, [Online]. Available: https://www.osti.gov/biblio/1488489.

[10] C. Spear, SystemVerilog for Verification: A Guide to Learning the Testbench Language Features, Springer Science & Business Media, 2008.

[11] PSL and SVA Assertion Languages, Springer Netherlands, Dordrecht, 2008, pp. 55–82, [Online]. Available: https://doi.org/10.1007/978-1-4020-8586-4_4.

[12] C. Dax, F. Klaedtke, M. Lange, On regular temporal logics with past, Acta Inform. 47 (4) (2010) 251–277, [Online]. Available: https://doi.org/10.1007/s00236-010-0118-3.

[13] Accellera Systems Initiative (Accellera), Universal verification methodology (UVM) 1.2 user's guide, 2015, https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf.

[14] M. Naylor, S. Moore, A generic synthesisable test bench, in: 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2015, http://dx.doi.org/10.1109/MEMCOD.2015.7340479.

[15] L. Truong, S. Herbst, R. Setaluri, M. Mann, R. Daly, K. Zhang, C. Donovick, D. Stanley, M. Horowitz, C. Barrett, P. Hanrahan, Fault: A python embedded domain-specific language for metaprogramming portable hardware verification components, in: S.K. Lahiri, C. Wang (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2020, pp. 403–414.

[16] C. Wolf, Yosys Open SYnthesis Suite, https://github.com/YosysHQ/yosys.

[17] C. Wolf, Design and Implementation of the Yosys Open SYnthesis Suite (Bachelor Thesis), Vienna University of Technology, 2013.

[18] R. Brayton, A. Mishchenko, ABC: An academic industrial-strength verification tool, in: T. Touili, B. Cook, P. Jackson (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 24–40.

[19] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, K. Sen, RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs, in: 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2018, pp. 1–8, http://dx.doi.org/10.1145/3240765.3240842.

[20] A. Dobis, T. Petersen, M. Schoeberl, Functional coverage-driven fuzzing for Chisel designs, in: Proceedings of the Fourth Workshop on Open-Source EDA Technology (WOSET), 2021.

[21] K. Ruep, D. Groß e, SpinalFuzz: Coverage-guided fuzzing for SpinalHDL designs, in: European Test Symposium, 2022.

[22] S. Deng, D. Gümüşoğlu, W. Xiong, S. Sari, Y.S. Gener, C. Lu, O. Demir, J. Szefer, SecChisel framework for security verification of secure processor architectures, in: Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy, 2019.

[23] L. De Moura, N. Bjø rner, Z3: An efficient SMT solver, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2008.

[24] S. Yu, Y. Dong, J. Liu, Y. Li, Z. Wu, D.N. Jansen, L. Zhang, CHA: Supporting SVA-like assertions in formal verification of Chisel programs (tool paper), in: International Conference on Software Engineering and Formal Methods, Springer, 2022.

[25] A.B. Mehta, Constrained random verification (CRV), in: ASIC/SoC Functional Design Verification: A Comprehensive Guide To Technologies and Methodologies, Springer International Publishing, Cham, 2018, pp. 65–74, [Online]. Available: https://doi.org/10.1007/978-3-319-59418-7_5.

[26] ARM, AMBA AXI and ACE protocol specification AXI3, AXI4, and AXI4-lite ACE and ACE-lite, 2011, https://developer.arm.com/documentation/ihi0022/e/.

[27] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, K. Asanović, Chisel: Constructing hardware in a scala embedded language, in: Design Automation Conference, 2012.

[28] K. Asanović, R. Avižienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, P. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, A. Waterman, The Rocket Chip Generator, Tech. Rep. UCB/EECS-2016-17, 2016.

[29] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, J. Bachrach, Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations, in: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2017, pp. 209–216, http://dx.doi.org/10.1109/ICCAD.2017.8203780.

[30] Veripool, Verilator, https://www.veripool.org/wiki/verilator.

[31] A. Dobis, H.J. Damsgaard, E. Tolotto, K. Hesse, T. Petersen, M. Schoeberl, Enabling coverage-based verification in Chisel, in: 27th IEEE European Test Symposium (ETS 2022); Conference Location: Barcelona, Spain; Conference Date: May 23-27, 2022; Conference lecture on May 25, 2022, 2022-05.

[32] K. Kuchcinski, R. Szymanek, JaCoP - Java constraint programming solver, in: CP Solvers: Modeling, Applications, Integration, and Standardization, co-located with the 19th International Conference on Principles and Practice of Constraint Programming ; Conference date: 16-09-2013, 2013.

[33] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, Y. Zhu, et al., Bounded model checking, Adv. Comput. 58 (2003).

[34] J. Marques-Silva, I. Lynce, S. Malik, Conflict-driven clause learning SAT solvers, in: Handbook of Satisfiability, 2021.

[35] C. Barrett, R. Sebastiani, S.A. Seshia, C. Tinelli, Satisfiability modulo theories, in: Handbook of Satisfiability, 2008.

[36] K.L. McMillan, The SMV system, in: Symbolic Model Checking, 1993.

[37] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, NuSMV: a new symbolic model checker, Int. J. Softw. Tools Technol. Transf. (2000).

[38] A. Mishchenko, et al., ABC: A system for sequential synthesis and verification, 2007, URL http://www.eecs.berkeley.edu/alanmi/abc.

[39] A. Niemetz, M. Preiner, C. Wolf, A. Biere, Btor2, BtorMC and boolector 3.0, in: International Conference on Computer Aided Verification, 2018.

[40] A. Biere, K. Heljanko, S. Wieringa, AIGER 1.9 and beyond, 2011.

[41] C. Wolf, SymbiYosys, [Online]. Available: https://github.com/YosysHQ/SymbiYosys.

[42] C. Barrett, P. Fontaine, C. Tinelli, The SMT-LIB Standard: Version 2.6, Tech. Rep., 2017, Available at www.SMT-LIB.org.

[43] IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012), 2018.

[44] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, et al., FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud, in: ISCA, 2018.

[45] Microchip, 2021, https://www.microchip.com/, Accessed: 2021-08-29.

[46] M. Eilers, P. Müller, Nagini: A static verifier for python, in: H. Chockler, G. Weissenbacher (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2018, pp. 596–603.

[47] P. Müller, M. Schwerhoff, A.J. Summers, Viper: A verification infrastructure for permission-based reasoning, in: B. Jobstmann, K.R.M. Leino (Eds.), Verification, Model Checking, and Abstract Interpretation (VMCAI), in: LNCS, vol. 9583, Springer-Verlag, 2016, pp. 41–62.

[48] V. Iyer, K. Laeufer, K. Sen, B. Nikolić, A high performance multi-threaded RTL testbench API, in: OSCAR22, 2022.
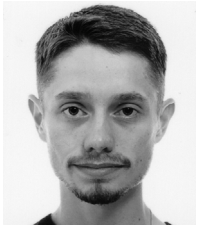
Andrew Dobis is a Masters student in Computer Science at ETH Zurich. After obtaining his B.Sc. in Computer Science from EPFL in 2020, he worked at the Technical University of Denmark (DTU)'s Embedded Systems Group as a Research Assistant. His research focused on improving the verification capabilities of the Chisel Hardware Construction language, which lead to the creation of the ChiselVerify verification library. His current research interests are more geared towards formal verification.

Kevin Laeufer is a Ph.D. candidate in the Department of Electrical Engineering and Computer Sciences at UC Berkeley. He received a B.Sc. degree in electrical engineering from RWTH Aachen University in 2015. He is interested in designing new languages, compilers and tools to automate programming, testing and design tasks. He is a major contributor to Chisel, in particular to the FIRRTL hardware compiler and the ChiselTest verification library.

Hans Jakob Damsgaard received the B.Sc. degree in electrical engineering in 2019 and the M.Sc. degree in computer science and engineering in 2021 as part of the Honours programme at the Technical University of Denmark, DTU. Currently, he is pursuing a Ph.D. on approximate reconfigurable accelerators for secure edge computing as part of the APROPOS project at Tampere University. His research interests include approximate hardware accelerators for neural networks, networks-on-chip, and computer architecture.

Tjark Petersen is a student at DTU in the final semester of the B.Sc. in electrical engineering. His research interest mainly revolves around digital design using Chisel, embedded systems and computer architecture.

Kasper Hesse is a Masters student in Computer Science and Engineering at the Technical University of Denmark (DTU). His work at DTU's Embedded Systems Group was mainly on exploring UVM and C-Scala co-simulation using the JNI.

Enrico Tolotto obtained an M.Sc. in Computer Science from the Technical University of Denmark in 2021. For his Masters thesis, Enrico worked on enabling constraint programming directly in Scala.

Simon Thye Andersen obtained an M.Sc. in Computer Science from the Technical University of Denmark in 2020. After graduating, he worked on converting VHDL to Verilog code to enable the use of VHDL with Chisel testbenches. He now currently works at Teledyne RESON as a Firmware and Hardware engineer.

Richard Lin obtained a Ph.D. in Computer Science from UC Berkeley in 2021. His research focuses on open-source electronics design with a programming languages and human–computer interaction focus, and he is a major contributor to Chisel and the ChiselTest framework. He is currently a post-doctoral researcher at UCLA.

Martin Schoeberl is Professor at the Technical University of Denmark (DTU), and has been since 2010. Before that, he was an Assistant Professor at the Institute of Computer Engineering at TU Vienna, where he had obtained his Ph.D. in 2005. His research interests are mainly in time-predictable computer architecture, real-time systems, and more recently hardware verification. His work on time-predictable architectures lead to the EC funded project T-CREST (Time-predictable Multi-Core Architecture for Embedded Systems). He is also a contributor to Chisel, and has been teaching Chisel in his digital design courses at DTU. He is the author of "Digital Design with Chisel".