Open-Source Formal Verification for Chisel

Kevin Laeufer, Jonathan Bachrach and Koushik Sen
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, USA
Email: {laeufer, jrb, ksen}@eecs.berkeley.edu

***Abstract*—Chisel is a Scala embedded hardware construction language allowing designers to take advantage of a general purpose programming language to generate digital circuit descriptions. From the beginning Chisel has featured integration with RTL simulators in order to allow designers to unit test their designs. We recently added support for easy formal verification of Chisel designs. Our implementation will be available to all users of the next Chisel and chiseltest releases. The source code is hosted on github and published under a permissive Apache 2 license.**

I. Introduction

While working on the RTL level description of a new circuit, designers need to quickly test their design in order to iteratively improve it. Extensive testing is also a common requirement before sending a design to be fabricated as bugs discovered after ASIC fabrication can be costly or even impossible to fix. The most common approach to testing RTL is to write a test bench program that interacts with a simulation of the design. Errors are found through manual waveform inspection or assertions in the design or the test bench.

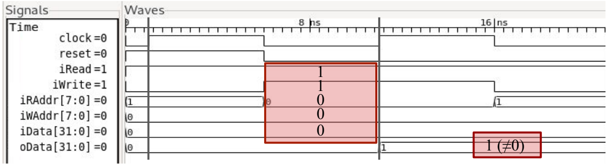
An alternative to exercising the circuit description with a set of concrete inputs is to symbolically explore the circuit execution for any inputs for a limited number of cycles. This technique is called bounded model checking [1] and works by unrolling the circuit for k cycles and then asking a SAT [2] or SMT [3] 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 [4], [5], [6]. 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 [7] 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 [8] or aiger [9] formats without having to worry about supporting the much more complicated Verilog standard. The open-source SymbiYosys [10] 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 [1].

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

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

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



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

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

Fig. 1. This example 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` (Section IV). It is based on a Verilog example from a popular blog. In the Chisel version, the assertion is automatically delayed until at least one cycle after reset, when there are valid `past` values available (Section VI). A bounded model check is executed by the `verify` command, which is called from a standard Scala unit test (Section III). 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.

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.

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

II. The Chisel Hardware Construction Language

Chisel is a modern hardware construction language embedded in the general purpose programming language Scala [11]. It allows designers to effectively write Scala programs that generate hardware descriptions at the register transfer level (RTL). One popular open-source application is the powerful RocketChip system on chip generator [12].

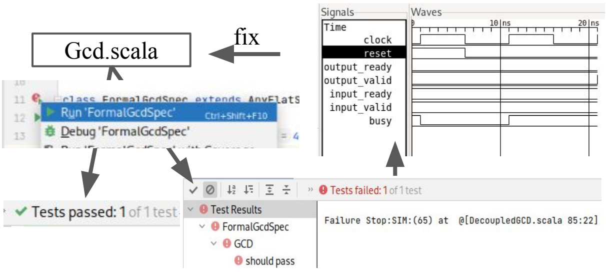
The user-facing API of Chisel is a Scala library with some syntactic sugar that allows the user to generate RTL designs. These designs then have to be converted into a format that is understood by simulators as well as FPGA and ASIC synthesis tools. The lowering is done by the FIRRTL compiler which converts a high-level intermediate representation (IR) into a normalized structural representation [13]. 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. The Chisel frontend makes it possible to attach annotations (i.e., meta-data) to arbitrary signals in the circuit and to schedule compiler passes to be executed that are free to consume the annotations and make changes to the circuit. We will make use of these facilities throughout the paper.

III. Our Formal Verification Flow

Before we dive into some of the details of our implementation we want to present the workflow that we imagine and illustrate how easy it can be to get started with formal verification of a Chisel circuit [2]. The recommended way to start a Chisel project is to use the open-source Chisel template repository [3]. The resulting Scala project automatically includes dependencies on the Chisel and the chiseltest libraries which will be downloaded by the Scala build tool.

The template contains an example of using the chiseltest library to test a greatest common denominator (GCD) circuit in simulation. This test can be executed through a Scala IDE or from a shell with the `sbt test` command. In order to turn this test into a formal check, we just need to substitute the `test(new DecoupledGcd(16))` command with `verify(new DecoupledGcd(16),` as well as provide the type of verification job as `BoundedCheck(10)` and extend the testing class with the `Formal` trait. 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 [14].



```
class FormalGcdSpec extends AnyFlatS
```

✓ Tests passed: 1 of 1 test

Tests failed: 1 of 1 test

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

Fig. 2. 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.

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, we can add assertions directly to the circuit by using the Chisel `assert` statement. The decoupled GCD circuit used as an example has an `input` and an `output` channel as well as a 1-bit `busy` register. We expect that while the circuit is busy, no new input is accepted:

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

This assertion will pass because the circuit does indeed fulfil 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 our 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))
}
```

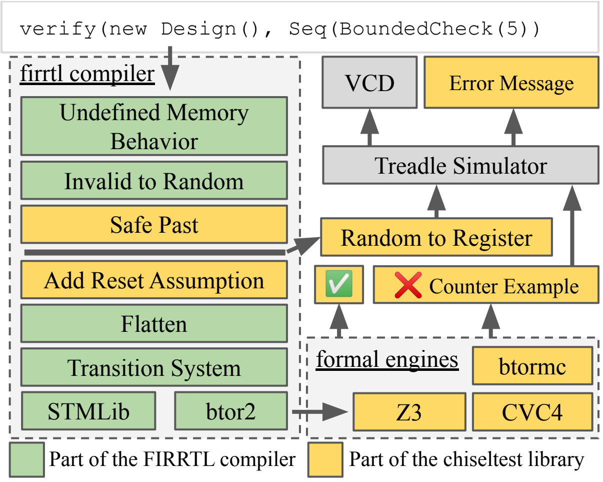
Here we make use of our `past` function for temporal properties which is described in detail in Section VI.

IV. A Formal Backend for FIRRTL

In order to implement the verify command introduced in the previous section, 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

[2] At the time of writing, some of the improvements discussed in this paper are pending pull requests. If you want to follow along with the examples in this paper, you can find them on our fork of chiseltest: https://github.com/ekiwi/chisel-testers2/tree/woset-examples. We are confident that by publication time all of them will be part of the upstream projects and we will be able to remove this footnote and point to a plain example repository using the published Scala libraries.

[3] https://github.com/freechipsproject/chisel-template



```
verify(new Design(), Seq(BoundedCheck(5)))
```

Fig. 3. The `verify` command is implemented as part of the chiseltest library and uses several compiler passes that make up the FIRRTL formal backend. We hook into the FIRRTL compiler to model undefined behavior with `DefRandom` statements and to delay temporal assertions as part of our safe past construct. We then add reset assumptions, flatten the system, convert to a formal transition system and then serialize the system to SMTLib or btor2. We provide bindings to launch various formal engines from chiseltest. If a counter example is found, we convert the `DefRandom` nodes in the circuit to registers before loading the circuit into the treadle simulator to replay the failure and obtain a simulation quality VCD and error message.

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 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 which 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. Figure 3 shows our compilation flow in more detail.

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 [15]. 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.

V. 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.

VI. 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 [16]. 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 Verlog 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 past 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 if 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 [17].

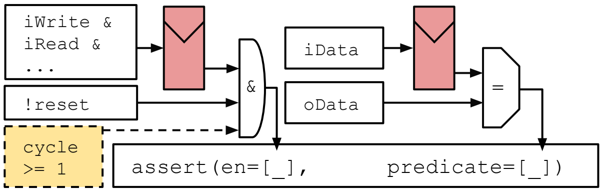


Fig. 4. The temporal assertion from Figure 1 results in a circuit with two registers created by the `past` function: One to delay the condition from the `when` statement and the other to delay the input data before it is compared to the current output data. By default an assertion is only enabled when 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.

VII. Conclusion

We introduced a new formal verification infrastructure for RTL designs written in Chisel. Since everything is integrated with our open-source FIRRTL compiler and chiseltest testing library, this support will be available to all Chisel users. In order to lower the barrier to entry, we added default reset assumptions and a safer version of the `past` function for temporal assertions. We carefully designed the formal backend of the FIRRTL compiler to model worst-case behaviors from the FIRRTL specification and to ensure that all counter examples can be replayed in simulation. We have ported several Verilog examples to our new Chisel formal verification infrastructure [4] and are looking forward to getting more feedback from our users.

Acknowledgment

We would like to express our thanks to the members of the Chisel community for their inspiration and help with making native formal verification for Chisel a reality. In particular we are grateful to Tom Alcorn, Daniel Kasza, Jack Koenig, Deborah Soung, Chick Markley, Schuyler Eldridge and Jiuyang Liu. We would also like to thank Clair Wolf for all she has done to advance the open-source Verilog ecosystem. Without yosys as an inspiration we would have never been able to conduct this work. This work was supported in part by Semiconductor Research Corporation. 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.

References

[1] A. Biere, A. Cimatti *et al.*, "Bounded model checking." *Advances in Computers*, vol. 58, 2003.

[2] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-Driven Clause Learning SAT Solvers," in *Handbook of Satisfiability*, 2021.

[3] C. Barrett, R. Sebastiani *et al.*, "Satisfiability Modulo Theories," in *Handbook of Satisfiability*, 2008.

[4] K. L. McMillan, "The SMV System," in *Symbolic Model Checking*, 1993.

[5] A. Cimatti, E. Clarke *et al.*, "NuSMV: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, 2000.

[6] A. Mishchenko *et al.*, "ABC: A System for Sequential Synthesis and Verification," *URL http://www. eecs. berkeley. edu/alanmi/abc*, 2007.

[7] C. Wolf and J. Glaser, "Yosys-a free Verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics*, 2013.

[8] A. Niemetz, M. Preiner *et al.*, "Btor2, BtorMC and Boolector 3.0," in *International Conference on Computer Aided Verification*, 2018.

[9] A. Biere, K. Heljanko, and S. Wieringa, "Aiger 1.9 and beyond," 2011.

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

[11] J. Bachrach, H. Vo *et al.*, "Chisel: Constructing Hardware in a Scala Embedded Language," in *Design Automation Conference*, 2012.

[12] K. Asanović, R. Avižienis *et al.*, "The Rocket Chip Generator," Tech. Rep. UCB/EECS-2016-17, 2016.

[13] A. Izraelevitz, J. Koenig *et al.*, "Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations," in *International Conference on Computer-Aided Design*, 2017.

[14] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[15] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Tech. Rep., 2017, available at `www.SMT-LIB.org`.

[16] "IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language," *IEEE Std. 1800*, 2017.

[17] S. Karandikar, H. Mao *et al.*, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," in *ISCA*, 2018.

[4] https://github.com/ekiwi/chisel-testers2/tree/woset-examples/src/test/scala/chiseltest/formal/examples