Improving the Verification Efficiency of Chisel Designs with Multi-Level Code Coverage

Andrew Dobis, Enrico Tolotto, Hans Jakob Damsgaard, Martin Schoeberl

Department of Applied Mathematics and Computer Science
Technical University of Denmark
Lyngby, Denmark

andrew.dobis@alumni.epfl.ch, s190057@student.dtu.dk, s163915@student.dtu.dk, masca@dtu.dk

***Abstract*—Ever-increasing performance demands are pushing hardware designers towards the use of domain-specific accelerators. This means that more and more hardware must be designed from scratch in an ever shortening amount of time. We must thus find a way to improve the overall efficiency of the hardware design and verification cycles. The design efficiency was improved with the introduction of Chisel, however we still need to improve the verification efficiency. That is why we propose, in this paper, multiple solutions for adding one of the most important verification tools there is: code coverage.**

We explored and created methods to be able to obtain Statement coverage of a Chisel design at multiple levels of abstraction: at the Verilog, FIRRTL and Scala levels. The Verilog statement coverage is obtained by hooking into verilator and extracting the coverage results. The FIRRTL statement coverage is done by modifying the Treadle execution engine and running a custom FIRRTL compiler pass that adds coverage-gathering outputs to the generated AST in order to obtain information about the executed paths. Finally the Scala coverage is obtained using software coverage gathering tools found in the ScalaTest library. After exploring the potential statement coverage solutions, we move on to presenting our own Scala-implemented solution for gathering functional coverage on a Chisel design.

We then finish off by presenting an industry-provided use case that illustrates the efficiency of our coverage tools. These solutions give the verification engineer ample ways to gather coverage on a Chisel design at any abstraction level. Note that a discussion could be had on the usefulness of coverage information at certain levels of abstraction (i.e. high-level coverage might not be interesting when using hardware generators), but for now this is left up for the reader to decide.

***Index Terms*—hardware verification, code coverage, SystemVerilog, Chisel, Scala, FIRRTL**

I. Introduction and Objectives

As time passes, contemporary hardware design is met with tighter and tighter time constraints. This is added to the fact that, with the halting of Moore's law, hardware designers are turning to domain-specific accelerators in order to keep up with the ever-increasing performance demands [**?**]. This means that more and more hardware must be designed from scratch in shorter and shorter time periods [**?**]. However, the most widely used hardware description languages (i.e. VHDL and Verilog) are completely outdated and inefficient. To solve said problem, researchers at the University of California in Berkeley proposed Chisel [**?**], a Scala embedded high-level hardware construction language.

This solution is great, but is still lacking in verification functionalities and one of the main tools needed for the verification of digital systems is Code Coverage. This allows verification engineers to measure their progress throughout the testing process and have an idea of how effective their tests actually are. Coverage can be separated into multiple distinct categories, but we will focus on the following two: Statement and Functional coverage. Statement Coverage defines a quantitative measure of the testing progress, *"How many lines of code have been tested?"*, whereas Functional Coverage gives a rather qualitative measure, *"How many functionalities have we tested?"* [**?**].

In this paper we will propose three different methods for obtaining statement coverage, each at different levels of abstraction. We will present these methods in a bottom-up fashion: first off we will talk about obtaining statement coverage of the Verilog description generated by Chisel, secondly we will show our solution for getting statement coverage of the FIRRTL intermediate representation and finally we will talk about getting coverage directly at the Scala level. Once that is done we will also present our solution for gathering functional code coverage of the Chisel design directly in Scala. In the following section we will take a brief look at the Chisel hardware construction language, how FIRRTL comes into play and gather the knowledge needed to fully appreciate our solutions.

II. Background and State-of-the-Art

We will begin our brief overview by presenting Chisel and all of the tools surrounding it.

a) Chisel: Chisel is a hardware construction language embedded in the functional and Object-Oriented Programming (OOP) language Scala [**?**]. This means that a Chisel design actually generates a Verilog description that can then be synthesised. The language itself has syntax rooted in Scala, since Chisel is technically Scala code. This thus allows the description of hardware in a high-level manner, which is thus much for efficient than traditional HDLs like VHDL or Verilog. Scala also allows for both functional and OOP

constructs, which makes it possible to organise a design implementation very intuitively using Scala classes and objects and also to use the power of higher order functions to greatly simplify descriptions thanks to constructs like *mapping* or *reductions*.

b) FIRRTL: In modern a Chisel design, the source code is first compiled into an intermediate representation that is used as a sort of "optimisation layer" before being converted into the final Verilog form. During this optimisation process the origin Chisel description goes through three different intermediate representation layers:

- High-FIRRTL, which is a form that maps perfectly back to chisel, but with the firrtl structure.
- Mid-FIRRTL, which is a form where abstract constructs are simplified, i.e. loops are unrolled and arrays are flattened.
- Low-FIRRTL, which maps perfectly to RTL code with high-level conditional statements turned into multiplexers.

Throughout this optimisation process, custom FIRRTL compiler passes, known as *Transforms* can be used to modify the design. This is often done when trying to apply simplifications to the design to make the generated hardware more optimal [**?**].

c) Code Coverage: In software development, code coverage is used as a metric to measure the completeness of a testing suite. In recent years, these techniques, originally used for software, have been brought into the hardware verification universe in an attempt to reduce post-printing errors. The coverage metric can be defined in multiple ways, but in this paper we will mostly focus on two key aspects of it: Statement Coverage and Functional Coverage.

Statement Coverage. When wanting to retrieve coverage information about a specific program or design implementation, statement coverage will give us a quantitative measure of our testing progress. It measures which percentage of individual statements in our code have been tested. This metric can be very useful in the software world, when it comes to completeness of a testing suite, however it is only a partial solution in the hardware world, since it only tells us which lines were tested and not how well they were tested.

Functional Coverage. If we are looking to measure the completeness of a test suite, we can not forget to ask the following question: "are we even testing the right thing?". In the hardware world, engineers usually implement designs based off of pre-determined specifications, so when testing, we should also have a metric for how well we are implementing the specification. This is where Functional Coverage comes in. The idea is to first define what is called a *Verification Plan* [**?**], which is supposed to represent the specification we are trying to implement. Once that is done, we can then sample the different points defined in our specification during the testing process, to obtain results in the form of: *"test suite T covered a total of x% of the values specified by point P in specification A"*. This was first implemented in SystemVerilog, in which a verification plan was defined using the following constructs:

- `Bin`: Defines a range of values that should be tested for (i.e. what values can we expect to get from a given port).
- `CoverPoint`: Defines a port that needs to be sampled in the coverage report. These are defined using a set of bins.
- `CoverGroup`: Defines a set of `CoverPoints` that need to be sampled at the same time.

Our solution tries to mimic this verification plan syntax to allow for a simplified transition for engineers in this field (this is discussed further in a later section).

d) State of coverage in Chisel: Up until recently, the main Chisel developers have mostly focused their energy on getting chisel up to the standards imposed by main stream HDLs like VHDL or Verilog. This means that verification features are mostly lacking from the Chisel package. Some testing frameworks like *ChiselTest* [**?**] have come to be, but they still lack coverage capabilities. All of this amounts to the realisation that if one wants to gather code coverage of a current Chisel design, we need to rely on basic Scala software code coverage tools (which we explore in more detail in a later section). This is why we have decided to focus our energy on this and to introduce code coverage solutions that are specifically tailored for Chisel and we will discuss this in great detail in the following sections.

III. Statement Coverage at the Verilog level

TODO: Enrico's work goes here

IV. Statement Coverage at the FIRRTL level with Treadle

The first part of our solution is about code coverage, more specifically line coverage that was added to the Treadle FIRRTL execution engine. Treadle is a common FIRRTL execution engine used to simulate designs implemented in Chisel. This engine runs on the FIRRTL intermediate representation code generated by a given Chisel implementation and allows one to run user-defined tests on the design using frameworks like *iotesters* or the more recent *testers2*. In our pursuit of creating a verification framework, we found that one way to obtain line coverage would be to have our framework run on an extended version of Treadle that was capable of keeping track of said information.

The solution that was used to implement line coverage was based off of a method presented by Ira. D. Baxter [**?**]. The idea is to add additional outputs for each multiplexer in the design. These new ports, which we will call *Coverage Validators*, are set depending on the paths taken by each multiplexer and that information is then gathered at the end of each test and maintained throughout a test suite. Once the testing is done, we used the outputs gathered from the *Coverage Validators* to check wether or not a certain multiplexer path was taken during the test, all of this resulting in a branch coverage percentage.

This was implemented in Treadle by creating a custom pass of the FIRRTL compiler that traverses the Abstract Syntax Tree (*AST*) and adds the wanted outputs and coverage expressions into the source tree. Once that is done, the `TreadleTester` samples those additional outputs every time the `expect` method is called and keeps track of their values

throughout a test suite. Finally it generates a Scala `case class` containing the following coverage information:

- The multiplexer path coverage percentage.
- The coverage Validator lines that were covered by a test.
- The modified LoFIRRTL source code in the form of a `List[String]`.

The `CoverageReport` case class can then be serialized, giving the following report:

```
COVERAGE: 50.0% of multiplexer paths tested
COVERAGE REPORT:

+ circuit Test_1 :
+   module Test_1 :
+     input io_a : UInt<1>
+     input io_b_0 : UInt<2>
+     input io_b_1 : UInt<2>
+     input clock : Clock
+     output io_cov_valid_0 : UInt<1>
+     output io_cov_valid_1 : UInt<1>
+     output out : UInt<2>
+
+     io_cov_valid_0 <= io_a
-     io_cov_valid_1 <= not(io_a)
+     out <= mux(io_a, io_b_0, io_b_1)
```

The example above is taken for a simple test, where we are only testing the path where `in_a` is 1. This means that, since we only have a single multiplexer, only half of our branches have been tested and we would thus want to add a test for the case where `in_a` is 0. The report can thus be interpreted as follows:

- "+" before a line, means that it was executed in at least one of the tests in the test suite.
- "-" before a line, means that it wasn't executed in any of the tests in the test suite.

Treadle thus allows us to obtain coverage at the FIRRTL level. A more interesting result would be if the FIRRTL line coverage would be mapped to the original Chisel source. This is possible but challenging, since Treadle only has access to the original source code through *Source locators* which map some of the FIRRTL lines back to Chisel. This means that the code can only be partially mapped and the remainder will have to be reconstructed using some smart guessing.

V. Statement Coverage at the Scala level

TODO: Hans' work goes here

VI. Functional Coverage directly in Scala

TODO: Adapt this section to fit into a discussion on coverage rather than a presentation of a framework. Functional Coverage is one of the principal tools used during the verification process, since it allows one to have a measurement of *"how much of the specification has been implemented correctly"*. A verification framework would thus not be complete without constructs allowing one to define a verification plan and retrieve a functional coverage report. The main language used for functional coverage is *SystemVerilog*, which is why our solution is based on the same syntax. There are three main components to defining a verification plan:

- `Bin`: Defines a range of values that should be tested for (i.e. what values can we expect to get from a given port).
- `CoverPoint`: Defines a port that needs to be sampled in the coverage report. These are defined using a set of bins.
- `CoverGroup`: Defines a set of `CoverPoints` that need to be sampled at the same time.

Using the above elements, one can define what's known as a verification plan, which tells the coverage reporter what ports need to be sampled in order to generate a report. In order to implement said elements in Scala we needed to be able to do the following:

- Define a verification plan (using constructs similar to `coverpoint` and `bins`).
- Sample DUT ports (for example by hooking into the *Chisel Testers2* framework).
- Keep track of bins to sampled value matches (using a sort of DataBase).
- Compile all of the results into a comprehensible Coverage Report.

Implementing these elements was done using a structure where we had a top-level element, known as our `Coverage Reporter` which allows the verification engineer to define a verification plan using the `register` method, which itself stores the `coverpoint` to `bin` mappings inside of our `CoverageDB`. Once the verification plan is defined, we can sample our ports using the `sample` method, which is done by hooking into *Chisel Testers2* in order to use its peeking capabilities. At the end of the test suite a functional coverage report can be generated using the `printReport` method, which shows us how many of the possible values, defined by our bin ranges, were obtained during the simulation.

```
1  val cr = new CoverageReporter
2  cr.register(
3      //Declare CoverPoints
4      //CoverPoint 1
5      CoverPoint(dut.io.accu , "accu",
6          Bins("lo10", 0 to 10)::
7          Bins("First100", 0 to 100)
8          ::Nil)::
9      //CoverPoint 2
10     CoverPoint(dut.io.test, "test",
11         Bins("testLo10", 0 to 10)
12         ::Nil)::
13     Nil,
14     //Declare cross points
15     Cross("accuAndTest", "accu", "test",
16         CrossBin("both1", 1 to 1, 1 to 1)
17         ::Nil)::
18     Nil)
```

The above code snippet is an example of how to define a verification plan using our coverage framework. The concepts are directly taken from `SystemVerilog`, so it should be accessible to anyone coming from there. One concept, that is used

in the example verification plan, which we haven't presented yet is the idea of *Cross Coverage* defined using the `Cross` construct. *Cross Coverage* allows one to specify coverage relations between CoverPoints. This means that a cross defined between, let's say, `coverpoint a` and `coverpoint b` will be used to gather information about when `a` and `b` had certain values simultaneously. Thus in example verification plan we are checking that `accu` and `test` take the value 1 at the same time.

Once our verification plan is defined, we need to decide when we want to sample our cover points. This means that at some point in our test, we have to tell our `CoverageReporter` to sample the values of all of the points defined in our verification plan. This can be done, in our example, simply by calling `cr.sample()` when we are ready to sample our points. Finally once our tests are done, we can ask for a coverage report by calling `cr.printReport()` which results in the following coverage report:

```
============== COVERAGE REPORT ==============
================ GROUP ID: 1 ================
COVER_POINT PORT NAME: accu
BIN lo10 COVERING Range 0 to 10 HAS 8 HIT(S)
BIN First100 COVERING Range 0 to 100 HAS 9 HIT(S)
=============================================
COVER_POINT PORT NAME: test
BIN testLo10 COVERING Range 0 to 10 HAS 8 HIT(S)
=============================================
CROSS_POINT accuAndTest FOR POINTS accu AND test
BIN both1 COVERING Range 1 to 1 CROSS Range 1 to 1
HAS 1 HIT(S)
=============================================
```

An other option would be, for example if we want to do automated constraint modifications depending on the current coverage, to generate the coverage as a Scala `case class` and then to use it's `binNcases` method to get numerical and reusable coverage results.

One final element that our framework offers is the possibility to gater delayed coverage relationships between two coverage points. The idea is similar to how a `Cross` works, but this time rather than sampling both points in the same cycle, we rather look at the relation between one point at the starting cycle and an other point sampled a given number of cycles later. This number of cycles is called the `delay` and there are currently three different ways to specify it:

- `Exactly` delay, means that a hit will only be considered if the second point is sampled in its range a given number of cycles after the first point was.
- `Eventually` delay, means that a hit will be considered if the second point is sampled in its range at any point within the following given number of cycles after the first point was.
- `Always` delay, means that a hit will be considered if the second point is sampled in its range during every cycle for a given number of cycles after the first point was sampled.

VII. Efficiency Evaluation

TODO: Maybe add a bit on Tjark's use case?

VIII. Related work

TODO: Talk about the Cover Construct from Chisel, maybe a bit on coverage with SystemVerilog and ofc link the work to ChiselVerify

IX. Conclusion

We propose multiple solutions for the gathering of coverage data in high-level chisel designs. Previously, one needed to rely on other languages such as SystemVerilog in order obtain coverage information during the verification of a Chisel design. This meant that testing either fully needed to be done in an other language or we needed to turn to multi-language testing which was very often cumbersome and time-consuming. With our solutions we are bringing the world one step closer to making Chisel a fully capable design and verification language. We now no longer need to rely on other languages in order to gather statement and functional coverage of our Chisel designs. An interesting discussion to be had is around the abstraction-level that should be used to gather coverage. One could justify that high-level coverage information lacks usefulness when using hardware generators since we would rather have information on the execution of the generated hardware rather than the generator itself, but of course this discussion is left up for the reader to have. We also proposed a solution for functional coverage entirely in Scala, this is actually a small part of a much broader research project name ChiselVerify, done on bringing most aspects of design verification to the Chisel ecosystemTODO: cite ChiselVerify somehow.

X. Acknowledgements

We would like to thank the researchers at Berkeley who are constantly working on making Chisel a more ubiquitous hardware construction language and for the feedback related to our work on adding coverage to Treadle. We would also like to thank the members of the embedded systems section at DTU for providing a constantly stimulating environment and InfinIT for funding the project.