Towards an Open-Source Verification Method with Chisel and Scala

Martin Schoeberl, Simon Thye Andersen,
Kasper Juul Hesse Rasmussen
Department of Applied Mathematics and Computer Science
Technical University of Denmark
Lyngby, Denmark
masca@dtu.dk, simon.thye@gmail.com,
s183735@student.dtu.dk

Richard Lin
Department of Electrical Engineering and Computer Sciences
UC Berkeley
Berkeley, CA
richard.lin@berkeley.edu

***Abstract*—Performance increase with general-purpose processors has come to a halt. We can no longer depend on Moore's Law to increase computing performance. The only way to achieve higher performance or lower energy consumption is by building domain-specific hardware accelerators. To efficiently design and verify those domain-specific accelerators, we need agile hardware development.**

This paper presents a combination of open-source tools for verifying circuits described in mixed languages. It builds on top of the Chisel hardware construction language and uses Scala to drive the verification. We also explore the testing strategy used in the Universal Verification Methodology (UVM) in the context of verifying hardware described in Chisel.

***Index Terms*—digital design, verification, object-oriented programming**

I. Introduction

We can no longer depend on Moore's Law to increase computing performance [6]. Performance increase with general-purpose processors has come to a halt. The only way to achieve higher performance or lower energy consumption is by building domain-specific hardware accelerators [5]. Furthermore, the production of a chip is costly. Therefore, it is essential to get the design right at the first tape-out. Thorough testing and verification of the design is mandatory.

To efficiently develop and verify those accelerators, we can learn from software development trends such as agile software development [4]. We believe that for the road ahead, we need to adapt to agile hardware development [8].

Until a few years, the two main design languages Verilog and VHDL, dominated the design and testing of digital circuits. However, both languages are decades behind modern languages for software development.

Recent advances with SystemVerilog and Chisel [3], [11] have brought object-oriented programming into the digital design and verification process. SystemVerilog, an extension of Verilog, adds object-oriented concepts for the non synthesizable verification code. Chisel is a "Hardware Construction Language", embedded in Scala, to describe digital circuits. Circuits described in Chisel can be tested and verified with a Chisel testing framework and Scala tests. Scala/Chisel brings object-oriented and functional programming into the world of digital design.

One of Chisel's primary benefits is its reliance on object-oriented principles for hardware description. Other classes may inherit this but with different implementations by defining a base class, such as adding floating-point operations to an ALU. As both the basic ALU and the FP-enabled ALU derive from the same base class, they can be used interchangeably. The object-oriented programming approach in Chisel allows for easy parametrized hardware generation, as seen in e.g., the RISC-V Rocket Chip [2], and is also used in the Chisel test described in Section V-B. While VHDL supports configurations and multiple architecture definitions, these require the designer to start from scratch when defining a new architecture, instead of simply improving or adding onto the existing architecture.

This paper describes a research project that aims to build a testing framework in Scala that takes the best methods from the Universal Verification Methodology (UVM) and decades of experience in software testing. Furthermore, we aim to build on open-source projects only. Therefore, our work is open-source as well.

The main contribution of this paper is the exploration of available open-source tools with a small example design. We can verify digital designs written in mixed languages such as Verilog, VHDL, and Chisel and simulate all of them in a tool-flow consisting of open-source tools only.

II. The Universal Verification Methodology

The Universal Verification Methodology (UVM) is a methodology for testing and verifying of digital circuits, introduced in 2011. Previously, verification methodologies were vendor-specific, forcing users to stay with one tool or to spend a lot of time and money transitioning to a new tool. UVM is unique in the fact that it is an Accellera standard developed together with all of the major EDA vendors, such as Questa, Cadence, and Synopsys. As of 2017, it has also been standardized as IEEE 1800.2. Although SystemVerilog and UVM are IEEE standards, the standards are freely available through the IEEE Get program [9].

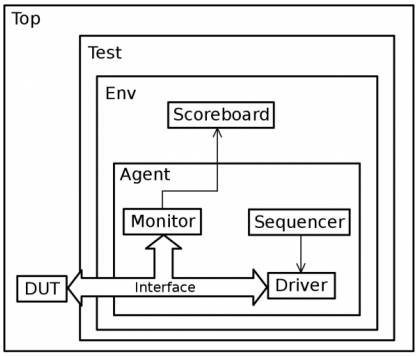

Fig. 1. Representation of a simple UVM testbench. By Pedro Araújo / colorlesscube.com

UVM is implemented as a SystemVerilog library and utilizes the fact that SystemVerilog uses object-oriented programming (OOP) when designing testbenches. Using OOP patterns such as inheritance and polymorphism, the verification engineer can design generic components that can be extended and modified to provide application-specific functionality.

A. A UVM Testbench

UVM Testbenches consists of several distinct components, as shown in Figure 1. Each component performs only one task in the testbench, allowing the engineer to make changes to some components without affecting others. For example, the sequencer is the component responsible for generating transactions for the DUT, whereas the driver is responsible for converting the transaction into pin-level wiggles, i.e., generating correct start/stop conditions and driving signals. If a new sequence of transactions is to be generated, only the sequencer is affected. Likewise, the sequencer does not care how the transactions are converted into pin-level signals—this is the sole responsibility of the driver. This distinction into several components results in a more structured testbench design as there are fewer dependencies than in a monolithic testbench.

The main components of a UVM testbench are as follows:

A *Sequence(r):* defines the order of transactions necessary for a given purpose, e.g., synchronization or reset sequence. The sequencer is responsible for transferring the transactions, defined by the sequence, to the driver.

A *Driver* converts transactions into pin-level signals and drives these signals onto the DUT.

An *Interface* is a SystemVerilog construct which allows the user to group related signals. A DUT may have several interfaces attached. The interface is used to avoid hooking directly into the DUT, making it easier to test multiple DUT versions.

A *Monitor* monitors all traffic on the interface, converting pin-level signals into transaction-level objects that can be operated on by other components, such as a coverage collector or scoreboard.

An *Agent* encapsulates monitor, sequencer and driver, setting configuration values. Agents may be set active or passive (with or without a driver and sequencer). An agent is useful when it is necessary to have multiple instances of the same components, e.g., when a 4-port network switch needs four identical drivers with different configurations.

A *Scoreboard* is used to check whether correct functionality is achieved. Usually does so by using a "golden model" for co-simulation via the SystemVerilog direct programming Interface.

The *Environment* is used to configure and instantiate all child components. Environments are typically application-specific and may be modified by the test.

The *Test* is the top-level verification component. The test designer may choose to perform factory overrides of classes and set configuration values here, which modify the child components.

As shown above, even a "Hello, World" example using the UVM requires that the user understands how and why each of the different UVM components should be used. The use of so many components causes UVM to have a very steep learning curve, which may discourage adoption. This also means that UVM is not the proper testing methodology for small designs or one-off tests due to the initial workload. However, once the initial setup of the testbench is finished for large and complex designs, generating new tests becomes easier.

III. Open-Source Tools

Our project plans to use mainly open-source tools, as we believe that only the open-source movement can lead to tools for agile hardware development and open libraries for IPs and verification components.

A. Chisel

Chisel is a hardware construction language embedded in Scala [3]. Chisel allows the user to write hardware generators in Scala, an object-oriented and functional language. For hardware generation and testing, the full Scala language and Scala and Java libraries are available. For example, we read in the string based schedules for a network-on-chip and convert them with a few lines of Scala code into a hardware table to drive the multiplexer of the router and the network interface.

Chisel is solely a hardware *construction* language, and thus all valid Chisel code maps to synthesizable hardware. By separating the hardware construction and hardware verification languages, it becomes impossible to write non-synthesizable hardware and in turn, speeds up the design process. As Scala and Java's full power is available to the verification engineer, the verification process is also made more efficient.

B. ChiselTest

While Chisel ultimately produces Verilog, which can be tested with industry-standard tools and processes, those generally force the user to pick between simple but limited (e.g., Verilog testbenches) or complex but powerful (e.g., UVM testbenches).

ChiselTest [10], a nonsynthesizable testing framework for Chisel, instead emphasizes on usability and simplicity while providing ways to scale up complexity.

Fundamentally, ChiselTest is a Scala library that provides access into the simulator through operations like poke (write value into circuit), peek (read value from circuit, into the test framework), and step (advance time). As such, tests written in ChiselTest are just Scala programs, imperative code that runs one line after the next. This structure uses the latest programming language developments that have been implemented into Scala and provides a clean and concise interface, unlike approaches that attempt to reinvent the wheel like UVM.

Furthermore, ChiselTest tries to enable testing best practices from software engineering. Its lightweight syntax encourages writing targeted unit tests by making small tests easy. Furthermore, a clear and clean test code also enables the test-as-documentation pattern, demonstrating a module's behavior from a temporal perspective.

C. Simulators

While Chisel designs can be simulated with any simulator that accepts Verilog input, there are trade-offs involved in choosing simulators. Commercial simulators require expensive licenses, while the open-source Verilator has a high time cost for compilation despite being efficient per-cycle. On the other hand, Treadle[1] is a simulator that operates at the level of Chisel's intermediate representation, FIRRTL[2]. Simulators like Treadle avoid the step of generating Verilog code and compiling from Verilog, which can vastly reduce the setup time for tests and efficiently run suites of many short tests.

Verilator has the benefit of compiling the Verilog code before simulating it. This is much faster compared to event-driven simulators but also limits the capabilities, as it only works on synchronous designs. Verilator claims to be on par or faster than the "Big 3" simulators on single thread. However, it also supports multi-threaded simulation, which can greatly improve simulation times for large designs [14].

D. Scala

The test environment and the driving code is written in Scala. Scala, with its compatibility with Java, has a very rich open-source library ecosystem. If you need a tool, e.g., an ELF file reader to load a binary, there will be a Java library available for it.

Furthermore, we can use all the testing libraries that have been developed for software development. A popular library is ScalaTest.[3] A Chisel tester can be embedded in a ScalaTest component, and a simple `sbt test` will execute all the tests.

IV. Integrating Legacy Languages

A verification method is only usable when it can handle mixed-source designs. This means a Scala driven method must be able to test components written in Verilog, VHDL, and SystemVerilog.

Chisel has support for black boxes, which allows the use of Verilog code within the Chisel design. Therefore, it is relatively easy to integrate Verilog components when wrapped into a black box. However, this forces Chisel to use Verilator instead of Treadle to run the simulation, impacting startup time.

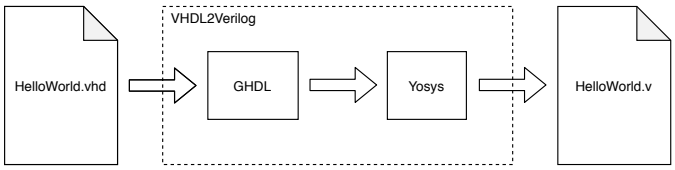

Fig. 2. VHDL2Verilog workflow

Chisel does not fully support VHDL. It can support VHDL using VCS, but there is no open-source solution available for VHDL simulation. For companies with a lot of source code written in VHDL this is a concern, as they must be able to integrate their existing IP in a Scala/Chisel based design and verification workflow. All major commercial simulation and synthesis tools support mixed-language designs, but no open-source tools exist that provide the same functionality.

To alleviate this issue, the open-source Yosys synthesis suite [15] can be used. Yosys is an open-source digital hardware synthesis suite for Verilog. Yosys also has a variety of plugins, one of these being a plugin for using GHDL [7], an open-source VHDL simulator. By using Yosys in conjunction with GHDL, VHDL files are compiled to an RTL-based intermediate representation, which is then written to a Verilog file using Yosys. GHDL has full support for IEEE 1076 VHDL 1987, 1993, 2002, and a subset of 2008. The workflow can be seen in Figure 2. A working solution named VHDL2Verilog has been made for this, which has been tested with certain simple VHDL designs [1].

Thus, using Yosys together with GHDL, it is possible to transpile VHDL to Verilog and then use a BlackBox to instantiate the generated Verilog code. This allows for an entirely open-source simulation toolchain, no matter which HDL has been initially used.

V. First Experiments

Although this is a work-in-progress report, we have started with an evaluation. We used an ALU with an accumulator from the Leros processor [12] as our device-under-test (DUT). The example is simple, but has a combinational part and state in a register, being a non-trivial circuit for testing.

The original design is in Chisel, and we reimplemented it in VHDL. We wrote tests in SystemVerilog/UVM and Scala. As execution platform, we used Synopsys VCS, ModelSim, Treadle, and Verilator.

We performed two experiments: (1) how to use hardware described in Chisel and VHDL in a UVM test setup and (2) how to test Verilog and VHDL components in a Chisel/Scala test setup.

A. Using UVM with Chisel

The Chisel toolchain translates Chisel code into plain Verilog for simulation and synthesis. Therefore, we can use a

[1]https://github.com/freechipsproject/treadle
[2]https://github.com/freechipsproject/firrtl
[3]https://www.scalatest.org/

UVM based test bench to test Chisel generated code. An important issue is that the modules and port names in the generated Verilog code are reasonable and do not change when changing the Chisel design.

We designed a UVM testbench to test the reference design's various implementations, the simple ALU from the Leros project. Constrained-random verification is performed by generating random stimulus and edge-case stimuli, and functional coverage is collected. Also, we use a scoreboard to verify the functionality of the DUT. We describe the reference model in SystemVerilog.

As a SystemVerilog interface connects the DUT to the testbench, no details about the DUT are exposed to the driver and monitor. This makes it easy to replace the DUT with an alternative implementation, e.g., in another HDL, and verify its functionality. It is only necessary to instantiate the new DUT and connect it to the interface, after which the test can be run.

We run the UVM testbench on the Verilog description generated by Chisel and on a VHDL version of the ALU. Using the VHDL version required more manual work to make the mixed-language simulation work, whereas the Verilog version was very fast to implement. Generating Verilog with Chisel and testing with UVM then proves to be a suitable workflow. However, no open-source SystemVerilog simulator with UVM support is known. Verilator primarily supports synthesizable constructs and does not support UVM, though this is on their roadmap [13].

B. Mixed Language Verification with Chisel

The initial experiment was to use UVM and test the VHDL and Chisel version of the design. From Chisel code, we can generate Verilog, which we can test with UVM. VCS and ModelSim support mixed language simulation (SystemVerilog and VHDL, but not Chisel). Therefore, this was straight forward.

However, for the open-source tool-flow, we start from the Chisel implementation. We wrote a test in Scala generating constraint-random values and comparing the DUT output with the output of a simulation written in Scala. The Chisel/Sala version can execute in Treadle for quick startup time or in Verilator for higher performance.

A Verilog implementation can be used in the Chisel setup by wrapping the DUT into a so-called `BlackBox`. When mixing Chisel with Verilog code, we need to use Verilator as a simulation engine. To reuse that Scala test with our VHDL implementation of the DUT, we use VHDL2Verilog to convert the VHDL version of the DUT to Verilog and wrap it into a BlackBox.

To reuse the test for two different implementations, we use the object-oriented features of Chisel/Scala. We define an abstract base class and extend that class by the Chisel implementation and the Chisel wrapper for the Verilog implementation. The tester expects the abstract base class. Using object-oriented programming to describe hardware is an exclusive feature of Chisel. SystemVerilog classes can only be used for test code, not to describe hardware.

In the end, we have a setup where we can use the full power of Scala (and Java) to test, co-simulate, and verify digital circuits described in Chisel, Verilog, or VHDL. This setup consists of open-source tools only.

C. The Road Ahead

This work-in-progress paper is a first sketch of the ideas to combine SystemVerilog/UVM with Chisel/Scala for a productive design and verification of future digital circuits. We will explore all combinations with a few more examples, provided by our industrial partners. From that, we will bootstrap adding constraint-random verification methods to Chisel testers and collecting coverage metrics within FIRRTL.

D. Source Access

As the project explores open-source tools for digital circuits design and verification, we provide all examples, including this paper, in open-source on GitHub:
https://github.com/chisel-uvm.

Acknowledgment

This work has been performed as part of the "InfinIT – Innovationsnetværk for IT", UFM case no. 1363-00036B, "High-Level Design and Verification of Digital Systems".

References

[1] Simon Andersen. Vhdl2verilog. https://github.com/chisel-uvm/vhdl2verilog.

[2] Krste et al. Asanovic. The Rocket Chip Generator. *EECS Department, University of California, Berkeley, Technical Report*, (UCB/EECS-2016-17), 2016.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In *The 49th Annual Design Automation Conference (DAC 2012)*, pages 1216–1225, San Francisco, CA, USA, June 2012. ACM.

[4] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. https://agilemanifesto.org/, 2001.

[5] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Commun. ACM*, 63(7):48–57, June 2020.

[6] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. Association for Computing Machinery.

[7] Tristan Gingold. Ghdl. https://github.com/ghdl/ghdl.

[8] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, January 2019.

[9] IEEE. 1800.2-2017 - IEEE Standard for Universal Verification Methodology Language Reference Manual.

[10] Richard Lin. ChiselTest. https://github.com/ucb-bar/chisel-testers2.

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

[12] Martin Schoeberl and Morten Borup Petersen. Leros: The return of the accumulator machine. In Martin Schoeberl, Thilo Pionteck, Sascha Uhrig, Jürgen Brehm, and Christian Hochberger, editors, *Architecture of Computing Systems - ARCS 2019 - 32nd International Conference, Proceedings*, pages 115–127. Springer, May 2019.

[13] Wilson Snyder. Verilator: Your Big 4th Simulator: Roadmap. https://www.veripool.org/papers/Verilator_Roadmap_CHIPS2019b.pdf, 2019.

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

[15] Clifford Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.