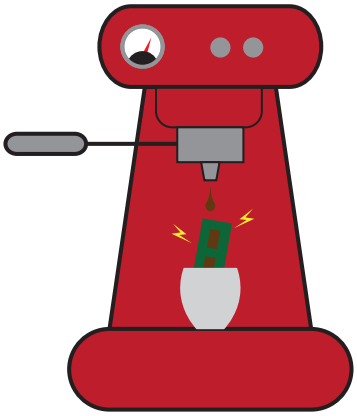


Verification of digital circuits using Java

02125 Bachelor Project

by
Rasmus Wiuff **s163977** s163977@dtu.dk

Supervisor
Martin Schoeberl masca@dtu.dk

Draft

Abstract

Add abstract

TODO LIST

Add abstract . . . 0
Document implementation . . . 5
Insert and describe tests on framework . . . 5
Write up conclusion . . . 5

CONTENTS

Page

1 Summary **2**

2 Introduction **2**
2.1 The UVM methodology . . . 2
2.2 Current verification tools . . . 2
2.3 The case against UVM and the mighty ABV . . . 2
2.4 The goal of this project . . . 3

3 Problem specification and analysis **3**
3.1 Simulation driver . . . 3
3.2 Peek-poke-step . . . 3
3.3 Assertion-logic . . . 3
3.4 Test translation . . . 3
3.5 Coroutines/concurrency . . . 3
3.6 Goals and success criteria . . . 4

4 Design **4**
4.1 Usecases . . . 4

5 Implementation **5**

6 Testing **5**

7 Conclusion **5**

References **6**

List of Figures **6**

List of Tables **6**

1 Summary

This section summarises the sections of the report.
Section 2 surmises UVM and some chosen chip verification frameworks, pivoting towards the motivation for, and introduction of, this project.
Section 3 breaks down the project into challenges and discusses solutions to these challenges.
Section 4 goes through the design considerations along with diagrams outlining the projects intended structure.
Section 5 describes the implementation and shows solutions of interest.
Section 6 explains the testing used to verify the project.
Section 7 summarises the project and discusses the results.

2 Introduction

In the world of designing integrated systems on chips, it is crucial to describe these in some hardware description language in order to test designs before actual manufacturing. Part of testing these designs include writing tests in some framework supporting some abstraction and then simulate the chip using the applied framework, a process known as *verification*. One verifies the *Device Under Test* or DUT. One of the most common methodologies is *Universal Verification Methodology*, or simply UVM.

2.1. The UVM methodology

Using UVM, one have to built testbench components. These include e.g. drivers for converting tests into proper DUT stimulus, monitors for reading the state of the DUT and scoreboards for comparing expected behaviour to actual behaviour, etc. UVM has the great benefit, that once these components have been defined, they can be reused within the scope of some system of designs. This means a high overhead, with high reusability.

2.2. Current verification tools

In today's landscape a myriad of verification tools exists. Projects like Chisel[1], and it's forked project SpinalHDL[2] discards the traditional hardware description languages and testbenches altogether and implements their own, and pyuvm[3] takes UVM into python using cocotb[4] as a backend driver. These tools do however have some problems. Chisel and SpinalHDL requires developers to learn a new description language, which means the communities of developers are small, increasing slowly. pyuvm uses Python which itself is an interpreted language, meaning its runtime resource requirements do not scale well on large projects.

2.3. The case against UVM and the mighty ABV

Chip verification is inherently done by hardware designers and engineers and UVM is inherently created by and for hardware designers. When software engineers verify their software, they use unit tests and assertions along with formal proofs. In recent years these approaches have been adopted by hardware designers. *Assertion Based Verification*, or ABV, along with formal verification is increasingly being applied to complex computing chip design verification and increases performance metrics over classic UVM[2]. Key impacts from this paper is summarised on Table 1.

Table 1: *Key impact of formal verification adoption over UVM[2]*

Verification cycles reduced by 25-30%
Pre-silicon bug detection rates improved by 20%
Security vulnerability detection increased by 40%

SystemVerilog Assertions, or SVA has been defined in IEEE 1800-2023[1, Chapter 16] meaning ABV is already a part of the SystemVerilog syntax.

[1] On GitHub: github.com/chipsalliance/chisel
[2] On GitHub: github.com/SpinalHDL/SpinalHDL
[3] On GitHub: github.com/pyuvm/pyuvm
[4] cocotb on the web: www.cocotb.org/

2.4. The goal of this project

The goal of this project is a framework written in a strongly typed language, supporting SystemVerilog and core ideas from ABV, thus making it easy for designers to write their designs in SystemVerilog and use a well known language to implement assertion based tests.
Introducing *SteelBrew*, the chip verification framework written in Java.

3 Problem specification and analysis

As mentioned in Section 2 this project aims to implement a testing framework for SystemVerilog designs, using core ideas of Assertion Based Verification, written in Java. This poses the following challenges:

- **Simulation driver:** The framework needs to communicate with some simulation driver.
- **Peek-poke-step:** Signal manipulation needs to be implemented to set up and carry out assertion of behaviour.
- **Assertion-logic:** ABV logic has to be implemented in the framework.
- **Test translation:** The framework has to translate test logic from Java to SystemVerilog and create a testbench for test simulation.
- **Coroutines/concurrency:** As assertions can be time-invariant, it makes sense to implement concurrency or coroutines to efficiently execute the simulation.

3.1. Simulation driver

Driving the simulations is not part of the scope of this project. Instead, the simulations are run by a third-party tool using Java to invoke the tool and listen for the results. Verilator[5] is a fast and community-driven simulator for SystemVerilog, which supports SVA directives. Through testbenches it is possible to set up and test DUT's fairly easily.

3.2. Peek-poke-step

Using the Java BigInt class, some logic for manipulating bits needs to be implemented in order to set up tests. BigInt enables describing integers in other bases, as is common in hardware design or low-level programming.

3.3. Assertion-logic

There are some assertion directives of interest:
1. Assert which raises an exception if some property does not hold.
2. Cover which monitors the coverage of some property.

These are the minimum that should be implemented. Some object for a test should be created and react if the driver raises an exception pertaining to the used directive. The logic in the framework needs to adhere to the logic stated in the SVA documentation.

3.4. Test translation

Manipulation and assertions has to be properly translated into a testbench that, when run, carries out the verification. This should be done in a manner that translates the test logic into a series of strings that are placed correctly within a testbench.

3.5. Coroutines/concurrency

To optimise runtime, it is desirable to run tests concurrently. For this purpose there are two options:
1. Java Runnable
2. Coroutines

[5] Verilator on the web: veripool.org/verilator

3.5.1. Java Runnable The Java Runnable interface[6] enables thread creation and joining. Benefits include the native support for Java and the documentation and community surrounding developing with Runnable. The main disadvantage is that threads rely on the operating system's scheduler, thus removing some degree of control during execution, and in the worst case, if not run in an elevated mode, threads might not get priority to run.

3.5.2. Coroutines Coroutines are described as "light threads" and enables a thread to control its own behaviour. This means threads can execute with a high degree of control. Disadvantages is that Java does not natively support coroutines. Solutions include hacking the JVM or using third party projects to execute coroutines. One such example is Javaflow[7] from Apache Commons or an interesting project called "Coroutines"[8] inspired by Javaflow.

3.6. Goals and success criteria

The discussed challenges and their solutions are listed on Table 2.

Table 2: *The projects challenges and success criteria*

#	Challenge	Success Criteria
1	Simulation driver	Logic that launches Verilator and handles communication with the processes.
2	Peek-poke-step	Logic that, using BigInt, manipulates wires and ports in the tests.
3	Assertion-logic	Logic that sets up assertion directives supported by the driver and SVA to support ABV.
4	Test-translation	Logic that translates the written test into a testbench that when executed, ensures the tests are carried out correctly.
5	Coroutines/concurrency	Implementation of concurrent execution of the simulations.

SteelBrew will be developed in an Agile way, based on use-cases. This ensures that a viable product is ready after dealing with challenge 1, 2 and 4 as these deals with setting up basic test functionality. Afterwards assertion logic, concluding with coroutines.

4 Design

The design of SteelBrew will take an Agile approach to get going with some framework for writing and running simple tests on devices written in SystemVerilog, using Java.

4.1. Usecases

Using SteelBrew should be as simple as possible. Fig. 1 shows an overview of the usecases for a developer. This is a simplification. As mentioned the framework is executed in Java, so to get started the user would start by creating a java project and importing SteelBrew. Hereafter the DUT is added and tests are defined and configured. A command should then execute the test and start the simulation.

4.1.1. Adding the device under test

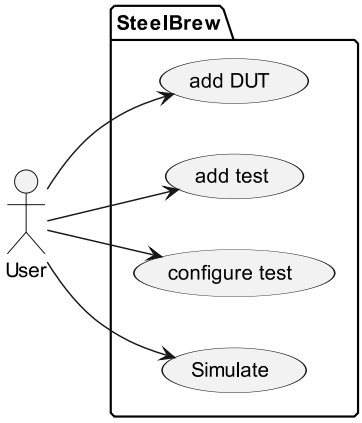
4.1.2. Adding tests

4.1.3. Configure tests

4.1.4. Simulate

[6] Documentation on Oracle: docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html
[7] Javaflow on Apache: commons.apache.org/sandbox/commons-javaflow/
[8] Coroutines on GitHub: github.com/offbynull/coroutines

Figure 1: *Usecase diagram for SteelBrew*



5 Implementation

Document implementation

6 Testing

Insert and describe tests on framework

7 Conclusion

Write up conclusion

References

[1] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". In: *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)* (Feb. 2024), pp. 1–1354. DOI: 10.1109/IEEESTD.2024.10458102. URL: https://ieeexplore.ieee.org/document/10458102.

[2] Kaushik Velapa Reddy. "Formal Verification with ABV : A Superior Alternative to UVM for Complex Computing Chips". In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* 10.6 (Nov. 5, 2024). Number: 6, pp. 90–98. ISSN: 2456-3307. DOI: 10.32628/CSEIT24106157. URL: https://ijsrcseit.com/index.php/home/article/view/CSEIT24106157.

List of Figures

1 Usecase diagram for SteelBrew 5

List of Tables

1 Key impact of formal verification adoption over UVM[2] 2
2 The projects challenges and success criteria 4

Listings