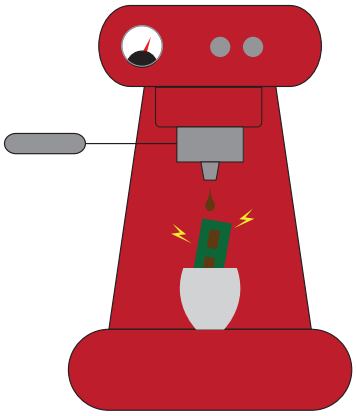


Verification of digital circuits using Java

02125 Bachelor Project

by
Rasmus Wiuff **s163977** s163977@dtu.dk
github.com/rwiuff/SteelBrew

Supervisor
Martin Schoeberl masca@dtu.dk

12th of May 2025

Abstract

This project investigates the possibility of using a custom framework developed in Java to create testbenches and run these using Verilator, to verify digital circuit designs. The project implements core verification testing techniques, and successfully manage to evaluate these using multithreading. The project investigates the possibility of using Assertion Based Verification using Verilator, however does not succeed in doing so. The report finalise with showcasing the projects workflow for simple verification tests and discusses further possible improvements.

Contents

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 has a look at the current landscape of chip verification that relates to this project.
Section 4 breaks down the project into challenges and discusses solutions to these challenges.
Section 5 goes through the design considerations along with diagrams outlining the projects intended structure.
Section 6 describes the implementation and shows solutions of interest.
Section 7 discusses the resulting project and its results.
Section 8 summarises the project and summarises 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. 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.

2.3. 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 Related works

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. As mentioned earlier ABV is increasingly becoming interesting as performance on verification of complex systems outperforms UVM[2]

4 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.

4.1. 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.

4.2. 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.

4.3. 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.

4.4. 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.

4.5. Coroutines/concurrency

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

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

4.5.1. 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.

4.5.2. 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.

4.6. 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.

5 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.

5.1. Use cases

Using SteelBrew should be as simple as possible. Fig. 1 shows an overview of the use cases 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.

5.1.1. Adding the device under test With the device design in the root folder, adding it to the framework should be as easy as adding the device to some object that handles device configuration. The underlying logic should take care of the rest.

5.1.2. Adding tests Since the tests are linked to some device, it is preferred that the same underlying logic ties tests and devices.

5.1.3. Configure tests This is the bulk of the interaction for the user. With some object carrying the tests and device, the user should be able to easily add new tests or assertions with some easy to use method calls. The underlying logic should resolve conflicts and handle test creation without the user getting much involved.

[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: *Use case diagram for SteelBrew*

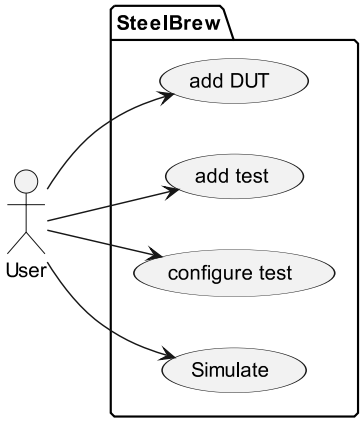
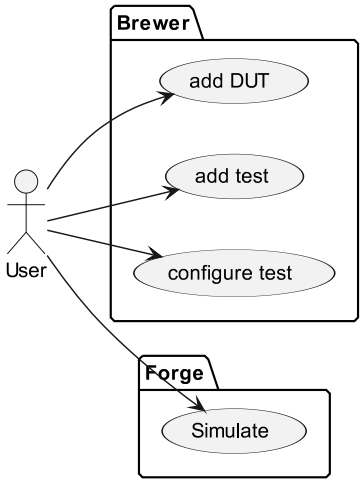


Figure 2: *The expanded use case diagram for SteelBrew*



5.1.4. Simulate Since the simulations are carried out by a third-party program, all handling between SteelBrew and said program, should be handled by yet another class, separate from the test-driven class. This ensures good separation of responsibility in the project.

5.2. Separation of responsibility

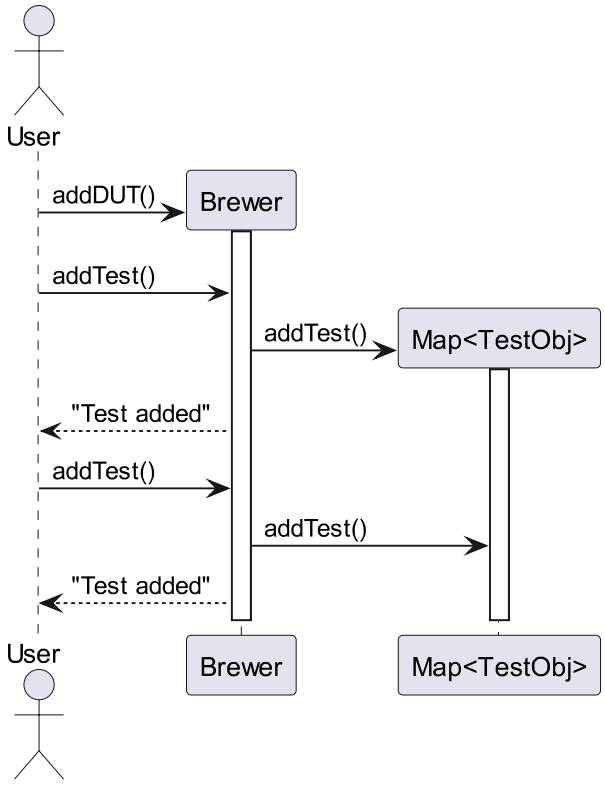
The project is considered two-fold:
1. Handling DUT's and tests
2. Handling concurrency and third-party software

For this reason the program has to contain two main components. The *Brewer* and the *Forge*. A third class is introduced as the entry-point for the program, and will be accessible to all other classes. This class is simply *SteelBrew*. Expanding on Fig. 1, the use case diagram is expanded as seen on Fig. 2.

5.3. The Brewer

The Brewer is the primary interface. There are several ways of going about this object handling multiple DUT's and tests. One aspect could be to hold Lists or Maps of the tests and devices, and then use methods to get these, and couple them accordingly. This has the advantage that from a schematic point of view it simplifies the program. However, once tests are applied to different devices, all contained within the same object, one has to keep track of internal links. The first major decision in the design phase was as such:
1. Give a Brewer to each DUT
2. Assign tests to each Brewer

Figure 3: *Sequence diagram showing the idea behind the Brewer*



This means more objects are created, but the Brewer object itself has a lot less responsibility in terms of tracking the internal links.
A sequence diagram on Fig. 3 shows this initial idea.

5.4. The Forge

The Brewers complement, the Forge handles another job, entirely. This class has to handle the responsibility of taking the tests from the Brewers and hand them down to the third-party tool. This prompts some considerations, namely how to launch other programs with very specific parameters and how to communicate with these programs.

5.4.1. Verilator and the Forge The third-party software chosen is Verilator. It is open-source and very quick at executing simulations. Verilator works like this:

1. The user prepares testbenches describing the testing environment and the tests
2. Verilator takes the testbench and some DUT as input and makes the files for compiling a C++ program
3. Compiling and running the program executes the tests and outputs a waveform-file, which can be inspected by e.g. GTKWave.[9]

This puts constrains on the design of the program. The Forge needs to call Verilator multiple times and then needs to run the compiled program. To ease this, the program creates a Makefile and executes through this. Simple redirects of stdout means that the Forge can collect the output from Verilator and the simulation.

5.4.2. Concurrency Up until now the main idea was to take a Brewer pointing to a DUT, add some tests and then hand it off to the Forge for Verilating, compiling and running the simulation. This poses one issue: This way of evaluating the program means that tests can only be parallised in terms of Verilator's internal multithreading, and then one DUT with all its test per thread. Verilator's internal multithreading is outside the scope of this project, so the only option is to get creative with the way tests are run. Normally when using Verilator, one testbench with all testing is run per DUT. The design choice here was to split up the tests, such that each test, or set of tests, gets its own testbench and its own compiled program. This way tests can be carried out in parallel by launching each workflow on different threads.

[9] GTKWave on the web: gtkwave.sourceforge.net

5.4.3. Coroutines It was of great interest to involve coroutines in the project. Java does not support coroutines natively, so workarounds were investigated. To this end, it turns out most coroutine frameworks and plugins are 6+ years old. They do not support newer Java versions, and thus not newer Java functionality. The design choice here was to use threads to execute tests.

5.5. Testbench

The workflow of Verilator and the design choice of splitting up tests into separate executables meant that the definition of tests were somewhat straightforward. The Brewer would have Testbench objects. Whenever tests are defined they are added to this object, and just before simulation, testbench files are created based on the content of the object. The testbenches have to have unique names, otherwise the Verilator executable and GNU Make will be unable to execute the correct benches.

5.6. Batch

While the Testbench class is able to contain all tests put in, it is basically a collection of strings. Furthermore, it is often desirable to control the sequence in which tests are executed, as the result from one test might affect the other. Or if one has to test what happens when manipulating signals inside the DUT. A new class is introduced to handle a sequence of tests, along with logic transforming simple methods into cumbersome strings for the testbench. The Batch class keeps a record of signals and tests involved in some scenario and when ready, creates a testbench with these tests executed as they are added by the user. The functionalities here starts with:

- **Peek** Read signal
- **Poke** Change signal
- **Step** Increment DUT clock

5.7. Assertions

The last leg of the project is assertion support. Assertions is a formal way of ensuring certain properties of a design, either in the scope of its entire execution or in certain scenarios. SystemVerilog has long supported assertions, however Verilator has limited support for assertions. For this reason some core functionality is designed and implemented, namely relations between signals in the DUT, using logic and relational operators.

5.8. Design overview

The workflow is illustrated by the sequence diagram shown on Fig. 4. The class diagram is shown on Fig. 5. Notice that the Forge is a singleton. This was chosen to make access to the Forge easier and most importantly ensure that only one Forge is instantiated. The Forge handles the executed threads, and multiple instances could lead to race conditions. The Barista class has been omitted from the diagram but will be discussed further in Section 5.

6 Implementation

In this section the practical implementation is showcased and discussed. Each class is briefly summarised, and interesting solutions are shown.

6.1. Build environment

The project exists in a Gradle[10] build environment consisting of an app and a library. The reasoning is simple: This project is developed as a library that the developer can import into their own Java code. The app is a merely a demo of how the library can be used, while the library itself is the project. If the project should be published Gradle will easily publish it to the Maven central repo. The demo app is in the `app` folder and the project is in the `steelbrew` folder. The root folder contains two ALUs written in SystemVerilog to experiment on.

6.2. SteelBrew

This class is the entry point for the project. It has two purposes. First it instantiates the Forge as a singleton. Second it has auxiliary functions for cleaning files produced during simulation.

[10] Gradle on the web: gradle.org

Figure 4: *Final workflow example as a sequence diagram*

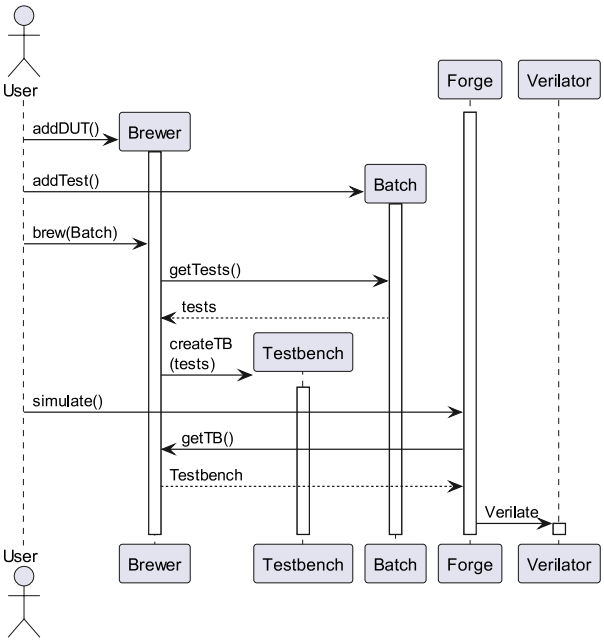
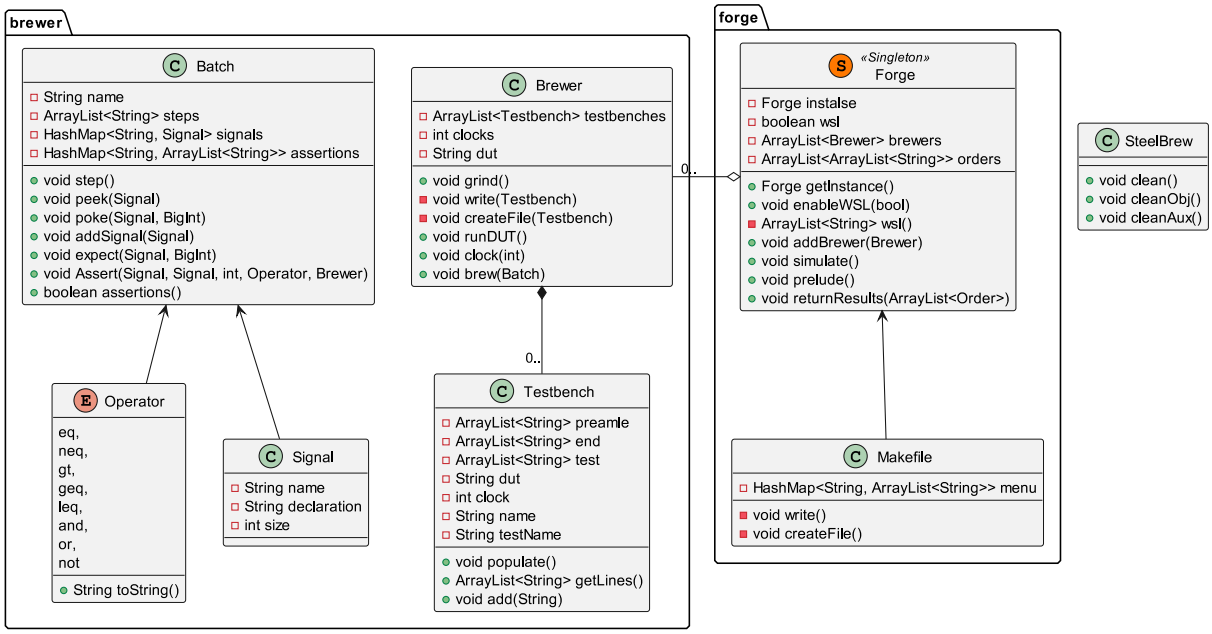


Figure 5: *Class diagram showing the structure of SteelBrew*



6.3. Brewer

The brewer class is constructed for each DUT to be tested. Upon construction, it adds itself to the Forge.
When the user is ready to simulate the Forge calls the `void grind()` method that creates and fills out testbench files. The `void brew(Batch batch)` method shows how a batch of tests is used to create a testbench. The method is shown on Listing 1. Listing 1 illustrates how tests are converted into testbench functions.

6.4. Testbench

The Testbench class holds all the necessary lines to make a testbench work. The `void add(String string)` method can then be used to add test lines to the testbench. The Brewer then assembles the file. Listing 2 shows how the

Listing 1: *Brewers brew method for creating a testbench from a batch of tests*

```
1         public void brew(Batch batch) {
2         Testbench testbench = new Testbench(dut, batch.getName(), clocks);
3         if (batch.assertions()) {
4             HashMap<String, ArrayList<String>> assertions = batch.getAssertions();
5             Set<String> functions = assertions.keySet();
6             functions.forEach(f -> assertions.get(f).forEach(s -> testbench.add(s)));
7             testbench.add("    while (sim_time < MAX_SIM_TIME) {\n");
8             testbench.add("        dut->clk ^= 1;\n");
9             testbench.add("        dut->eval();\n");
10            testbench.add("        if(dut->clk == 1){\n");
11            testbench.add("            posedge_cnt++;\n");
12            testbench.add("        }\n");
13            functions.forEach(f -> testbench.add(f+"\n"));
14            testbench.add("        m_trace->dump(sim_time);\n");
15            testbench.add("        sim_time++;\n");
16            testbench.add("    }\n");
17            testbench.add("\n");
18        }
19        ArrayList<String> steps = batch.getSteps();
20        for (String step : steps) {
21            testbench.add(step);
22        }
23        testbenches.add(testbench);
24    }
```

Listing 2: *Example of inserting custom names into the testbench*

```
1         preamble.add("int main(int argc, char** argv, char** env) {\n");
2         preamble.add("    V" + testName + " *dut = new V" + testName + ";\n");
```

Listing 3: *The peek method*

```
1     public void peek(Signal signal) {
2         steps.add("std::cout
          ↪  << \"\\n Peek on " + signal.getName() + ": \" << (int)(dut->" + signal.getName()
3                 + ") << \" @ simtime: \" << sim_time << std::endl;\n");
4     }
```

testbench names are assigned. When a testbench is created the variable `testName` combines the name of the DUT, with the name of the test.

6.5. Batch

The Batch class defines the test methods `step()`, `peek()`, `poke()`, `expect()` and `Assert()`. The peek method is interesting here, as it writes to stdout with its result. This can be seen on Listing 3. The assert method is also of interest. It takes two signals and an operator and then constructs a C++ method, that looks for some comparison, depending on the chosen operator. In Listing 1 the assertions are put into a testbench. The values of the assertion HashMap contains the method definition. The key is the function call. In Listing 1 the methods are defined early in the testbench and the call is the placed in a loop that evaluates every clock cycle, thus the assertion has to hold throughout the DUT evaluation. The two helper classes Operator and Signal will briefly be explained.

6.5.1. Operator The operator is a Java enum that comprises all the operators for comparison. The enum has a `toString()` method. If an assertion is added as such,
`batch.Assert(signal, signal2, 1, Operator.neq, alu);` the assert method uses the toString method to convert `neq` to "! =" when inserted in the C++ method.

Listing 4: *The write method that creates the recipes for the verilation, compilation and execution of simulations*

```
1     private void write() {
2         try {
3             FileWriter writer = new FileWriter("Makefile");
4             Set<String> duts = menu.keySet();
5             for (String dut : duts) {
6                 ArrayList<String> tests = menu.get(dut);
7                 for (String test : tests) {
8                     writer.write(".PHONY: " + test + "\n");
9                     writer.write("\n");
10                    writer.write(test + ": waveform" + test + ".vcd\n");
11                    writer.write("\n");
12                }
13                for (String test : tests) {
14                    writer.write("waveform" + test + ".vcd: ./obj_dir/V" + test + "\n");
15                    writer.write("\t@./obj_dir/V" + test + "\n");
16                    writer.write("\n");
17                    writer.write("./obj_dir/V" + test + ": .stamp." + test + ".verilate\n");
18                    writer.write("\t@echo \"### Building executable ###\"\n");
19                    writer.write("\tmake -C obj_dir -f V" + test + ".mk V" + test + "\n");
20                    writer.write("\n");
21                    writer.write(".stamp."
                          ↪  + test + ".verilate: " + dut + ".sv " + "tb_" + test + ".cpp\n");
22                    writer.write("\t@echo \"### VERILATING ###\"\n");
23                    writer.write("\tverilator
                          ↪  --trace -cc " + dut + ".sv --exe tb_" + test + ".cpp --prefix V"
24                                + test + "\n");
25                    writer.write("\t@touch .stamp." + test + ".verilate\n");
26                    writer.write("\n");
27                }
28            }
29            writer.write(".PHONY: clean\n");
30            writer.write("clean:\n");
31            writer.write("\trm -rf .stamp.*;\n");
32            writer.write("\trm -rf ./obj_dir\n");
33            writer.write("\trm -rf *.vcd\n");
34            writer.close();
35        } catch (IOException e) {
36            e.printStackTrace();
37        }
38    }
```

6.5.2. Signal The Signal class is a POJO[11] that defines the signal with a name, signal size and declaration, when used as a variable in C++.

6.6. Makefile

The Makefile class is used to create a GNU Make Makefile to simplify process calls from the Forger. This can be seen on Listing 4. Lines 8-11 gives the test recipe a name, and calls for a waveform creation. 14-26 then recursively goes through the recipes:

1. The waveform depends on the executable.
2. The executable depends on the verilation stamp (an empty file telling Make that verilation is not necessary on repeated makes).
3. The stamp then depends on verilating the testbench and DUT.

6.7. Forge

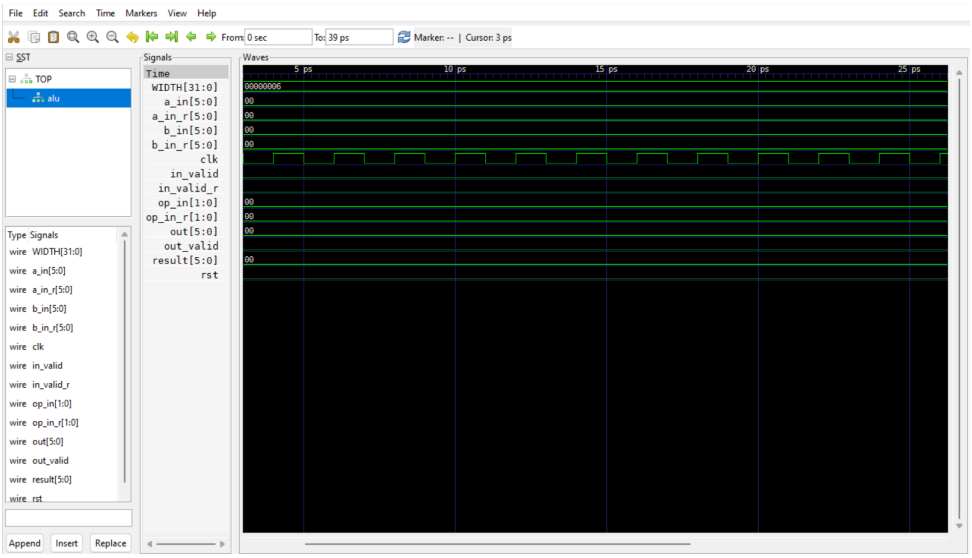
The Forge is where processes are handled. It is instantiated as a singleton, which means it is always accessible, and only one copy exists, making race-conditions less likely. The constructor also allows for lazy initialisation, meaning

[11] **P**lain **O**ld **J**ava **O**bject

Listing 5: *Running an ALU for 40 cycles*

```
1 SteelBrew steelBrew = new SteelBrew();
2 steelBrew.clean();
3 Forge.enableWSL(true);
4 Brewer alu = new Brewer("alu");
5 alu.clocks(40);
6 alu.runDUT();
7 Forge.simulate();
```

Figure 6: *Waveform after running the ALU for 40 cycles*



it is initialised when the `Forge.getInstance()` method is first called.
Forge has a `wsl()` method that solves the problem that Verilator does not ship binaries for Windows. However, if Windows Subsystem for Linux is installed with Verilator, Verilator can be called from a Windows terminal with the command `wsl -d Ubuntu <Command>`. The Forge executes simulations by calling `prelude()` which prompts the brewers to create testbench files. Afterwards the method creates an instance of Makefile and collects a list of all tests to be executed. The lists contain the command call for the process to start. Forge then constructs an instance of the Barista class, which in turn deals with a multithreaded approach.

6.7.1. Barista The Barista class is a helper class for the Forge class, that creates string builders that in turn executes the "make" command and saves stdout and stderr. Barista runs a test on its own thread, enabling concurrent simulation. Upon completion Barista calls `Forge.returnResults()`, which then prints the output of the tests, thus completing the simulations.

7 THE FINAL PRODUCT

This section will demonstrate the resulting workflow of the project for some various setups.

7.1. RUNNING THE DEVICE

The simplest workflow is running a DUT for some number of cycles. Consider Listing 5. Line 1 initialises SteelBrew and the Forge. Hereafter the root folder is cleaned for leftover files from previous simulations. Line 3 enables the Windows Subsystem for Linux support. On line 4 the ALU is added as the DUT. Line 5 sets the number of running cycles to 40 and 6 creates a testbench that simply runs the DUT for 40 cycles. On line 7 the program begins the simulation. The resulting file `waveformalurun.vcd` can then be inspected. The resulting screenshot is shown on Fig. 6. As seen from the figure, the ALU ran from 0 to 39 ps and the `clk` signal is behaving as expected. The simplest case works.

Listing 6: *Peek, poke, and step used on an ALU*

```
1          SteelBrew steelBrew = new SteelBrew();
2          steelBrew.clean();
3          Forge.enableWSL(true);
4          Brewer alu = new Brewer("alu");
5          Batch batch = new Batch("PeekPokeStep");
6          Signal signal = new Signal("in_valid", 1);
7          batch.addSignal(signal);
8          batch.peek(signal);
9          batch.poke(signal, BigInteger.ONE);
10         batch.step();
11         batch.peek(signal);
12         batch.step();
13         batch.poke(signal, BigInteger.ZERO);
14         batch.step();
15         batch.peek(signal);
16         batch.step();
17         batch.step();
18         alu.brew(batch);
19         Forge.simulate();
```

Figure 7: *Console output after using peek, poke and step on the ALU*

```
Peek on in_valid: 0 @ simtime: 1

Peek on in_valid: 1 @ simtime: 2

Peek on in_valid: 0 @ simtime: 4

################################
```

Figure 8: *Waveform changes with and poke*

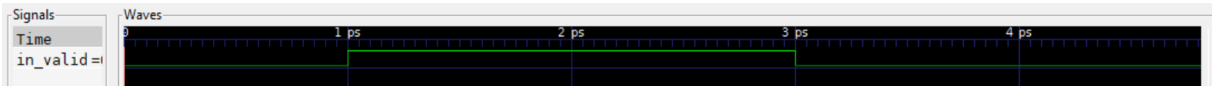


Figure 9: *Console output after using expect on the ALU*

```
Exptected 1 on in_valid. Recieved 0 @ simtime: 2

Peek on in_valid: 0 @ simtime: 2
```

7.2. Peek, poke and step

For the next workflow, the listing is shown on Listing 6. Lines 1-5 is seen in the previous example. On line 6 the variable `signal` is pointed to "`in_valid`", and on line 7 the signal is added to the batch. Lines 8–15 peeks at the signal, changes it to 1, peeks and changes it to 0, with some clock steps in-between. 16-17 starts the simulation. Fig. 7 shows the resulting console output and Fig. 8 shows the signal. The wave and the console output behaves as expected.

7.3. Expect

This next workflow demonstrate how to look for expected values. Consider Listing 7. The new addition here is line 9. We expect the signal to be a 1. Fig. 9 shows the console output. Exactly as expected.

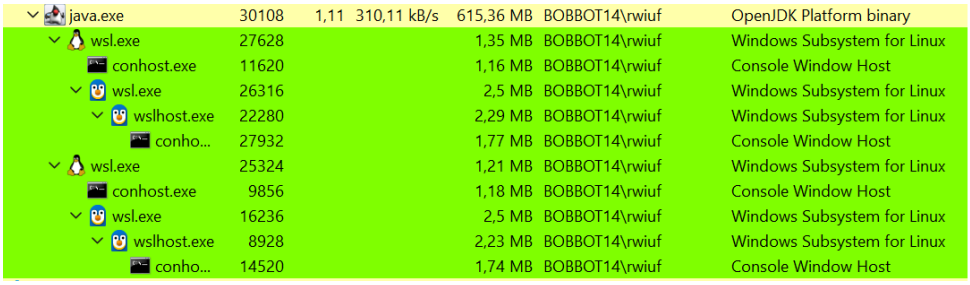
7.4. Assertions

Regrettably, the assertion methods did not work as expected. After rigorous experimentation, it would seem like the approach employed in the Assert method is inherently flawed. Placing the while loop in the testbench is needed for the current assertion-logic, however it rules out the possibility for peek and poke during a batch containing assertions. Further investigation is needed. Perhaps some randomised initial conditions combined with the expect method, would stand in for assertions, but this is truly not the way either.

Listing 7: *Expecting signal in the ALU*

```
SteelBrew steelBrew = new SteelBrew();
steelBrew.clean();
Forge.enableWSL(true);
Brewer alu = new Brewer("alu");
Batch batch = new Batch("Expect");
Signal signal = new Signal("in_valid", 1);
batch.addSignal(signal);
batch.step();
batch.expect(signal, BigInteger.ONE);
batch.peek(signal);
batch.step();
alu.brew(batch);
Forge.simulate();
```

Figure 10: *Multiple instances of Verilator created by Barista*



7.5. Concurrency

When executing multiple batches, the forge successfully runs multiple threads as seen on Fig. 10. This combined with Verilator's lightweight approach, and possibly, by employing its multithreading capabilities could dramatically speed up the verification process.

7.6. Further development

The groundwork is laid for further development. The development tasks would be as follows:
1. Implement working Assertions.
2. Employ Verilator multithreading.
3. Expand assertions with assume and cover logic.

The project as-is would be easily extendable with other testing capabilities, and as it is written in Java, the user could use their own logic, to prepare and execute tests.

8 Conclusion

The project did not reach its final goal: Implementing Assertion Based Verification.
However, other goals were reached. SteelBrew communicates with Verilator and using the Makefile, rather efficiently. The basic manipulation methods works as expected making the project employable as of this moment. The project further translates the tests in a way that makes parallel execution doable. The concurrency works as expected, and if more tests are added to the batches, this is a tool that can verify multiple DUTs in multiple setups simultaneously.

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

List of Tables

Listings