Automated testbench generation from digital timing information

Winand Seldeslachts

Supervisors: Prof. Luc Colman, Ir. Hendrik Eeckhaut (Sigasi nv)
Counsellor: dhr. Lieven Lemiengre (Sigasi nv)

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de industriële wetenschappen: elektronica-ICT

Department of Information Technology
Chair: Prof. dr. ir. Daniël De Zutter
Faculty of Engineering and Architecture
Academic year 2015-2016

Automated testbench generation from digital timing information

University of Ghent

Seldeslachts Winand

May 27, 2016

Permission for usage

The author gives permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use.

In the case of any other use, the limitations of the copyright have to be respected, in particular with regard to the obligation to state expressly the source when quoting results from this master dissertation

Winand Seldeslachts, 1 juni 2016

Acknowledgements

In the summer of 2015 I had the chance to start an internship for four weeks. The only problem was: I had to find a company that was willing to accept an intern by myself. It is then that I, through my promoter prof. Luc Colman, found and contacted Sigasi nv. They offered me an internship where I would write unit tests for software they were developing. After a very interesting month where I learned a lot, they made me an even more interesting offer. They were prepared to guide me during my masters thesis and proposed the topic discussed in this report. I was happy to accept and am very grateful for the opportunity they have given me. For this, and for their support and guidance during my graduation year I would like to thank the people of Sigasi nv, especially my supervisors dr. ir. Hendrik Eeckhaut and dr. ir. Lieven Lemiengre.

I would also like to thank professor Luc Colman for providing me with feedback and practical information. Also I am grateful to everyone who has helped me by proof reading this report.

Winand

Automated testbench generation from digital timing diagrams

Winand Seldeslachts

Supervisor(s): Prof. Luc Colman, Ir. Hendrik Eekhout, Ir. Lieven Lemiengre

***Abstract*: This report introduces a way to test VHDL designs in an easy and unambiguous way. A user describes the input signals and expected output signals for a specific design in a waveJSON [1] formatted file, the source file. WaveDrom [2] converts this file into so called documentation wave traces. These are the visual representation of what is described in the source file. At the same time the source file is converted into a VHDL test that will drive the design with the desired input signals and check the output signals. The test is simulated using the VUnit [7] framework. Simulating this test will result in so called simulation wave traces. These are compared to the documentation wave traces. The discrepancies are shown to the user in a clear way.**

***Keywords*: VHDL, WaveDrom, VUnit, automated verification, wave trace analysis**

I. Introduction

Verifying a design is an important step in designing a hardware system. Verification allows the designer to find and understand design flaws that compilers do not check. The traditional verification cycle for a hardware design is illustrated below.

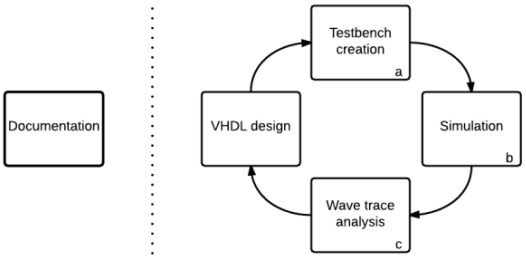

Figure 1: traditional verification cycle.

When a design has to be verified, **a)** A testbench containing one or more tests has to be created, **b)** These tests have to be simulated and **c)** the simulation results have to be analyzed. This analysis is done by comparing the resulting wave traces to the ones specified in the documentation.

This report introduces a way to streamline the verification of VHDL designs using a source file; a waveJSON formatted file. WaveJSON is an application of the JSON format. The purpose of WaveJSON is to provide a compact exchange format for digital timing diagrams. The software that has been built uses this file as a base for optimizing several steps in the verification process. The improved verification cycle is shown in figure 2.

W. Seldeslachts is an electronics engineering student, Ghent University (UGent), Gent, Belgium. E-mail: winand.seldeslachts@ugent.be

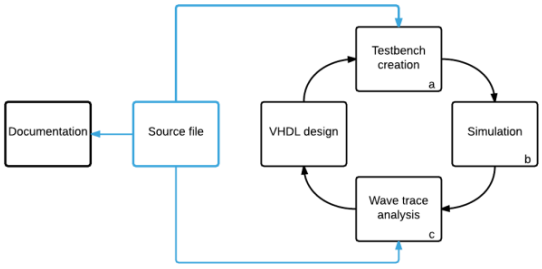

Figure 2: Improved verification cycle.

II. Problem analysis

The main issue with the traditional verification cycle is that test (contained in a testbench) are often outdated and not self-checking. Self-checking tests are tests that will automatically fail if the design does not meet the required specifications. This is opposed to tests where input signals are driven and the output results shown, but any conclusion regarding passing or failing is left to the designer, who has to visually check for compliance with the documentation. Using these outdated testbenches often results in staring contest between the designer and a bunch of wave traces.

Even if tests are self-checking, there is often no way to easily see where the error originated from.

These problems lead to the following research questions, which will be answered in this paper:

Can we design a system that will streamline the process of design validation?

Can we create self-checking testbenches with this system?

Can we optimize wave trace analysis with this system?

Does this system provide any benefits over existing verification methods?

III. Functional analysis

In an attempt to answer these questions, a software tool was developed using the Python programming language. This tool has two major tasks: build self-checking testbenches and analyze the wave traces. These are two steps in the verification cycle. The third step, simulation is handled by an external simulation engine. To do this, the VUnit framework is used. The VUnit framework extends the functionality of VHDL and makes it possible to start a simulator with a Python command. The function of the tool is illustrated in figure 3.

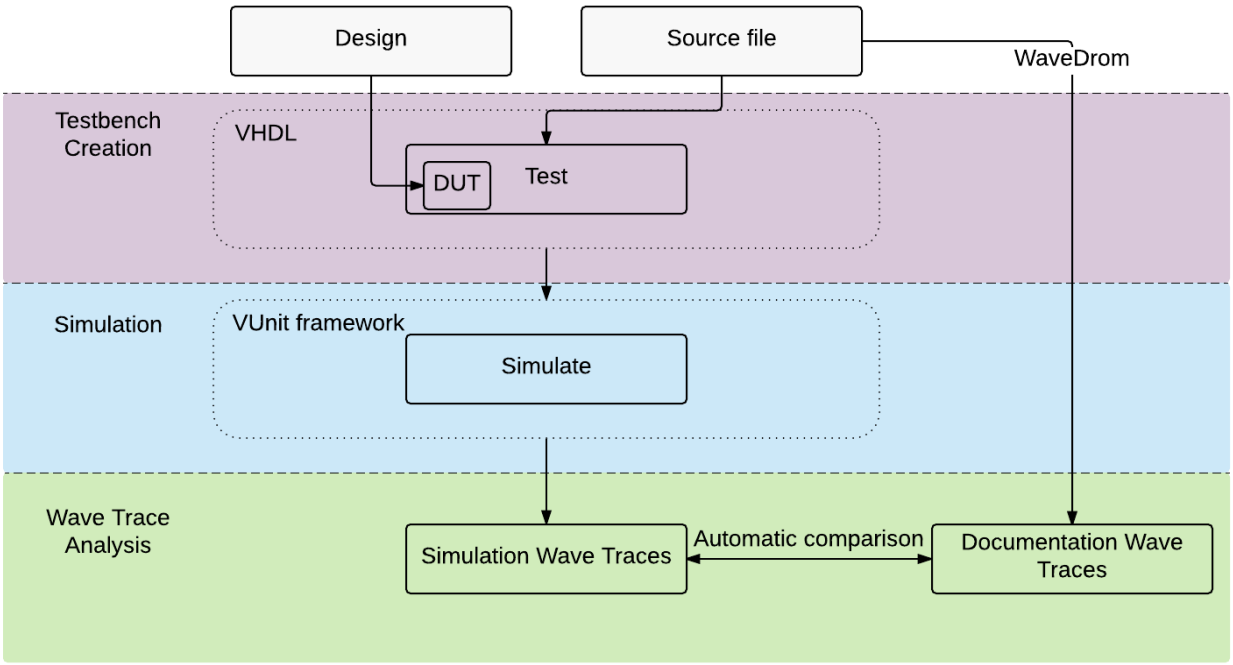

Figure 3: overview.

The testbenches are built according the VUnit requirements based on a template. This template defines the basic structure of the testbench. Using string manipulation the tool adds test specific code to this testbench based on the description in the source file. The result is a self-checking VHDL testbench for the targeted design.

At the same time the documentation wave traces are generated by WaveDrom. WaveDrom is an open source engine for generating wave traces from waveJSON files. It uses the description in the source file to visualize how the design should behave.

Then the test is simulated using the VUnit framework. This results in the simulation wave traces. These are compared to the documentation wave traces. If they are equal, the test has succeeded. If they are not, the difference is shown.

In short the software will convert a source file describing a feature of the design into a VHDL test, simulate it, and then output the comparison of simulation and documentation wave traces to the users as shown in figure 4.

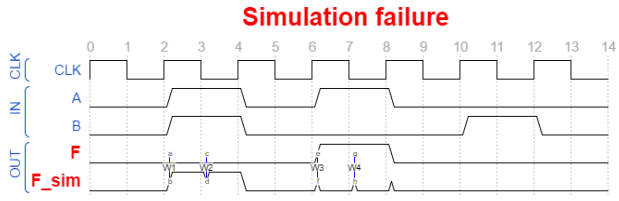

Figure 4: Output example.

This all is controlled by the graphical user interface (GUI) shown in figure 5.

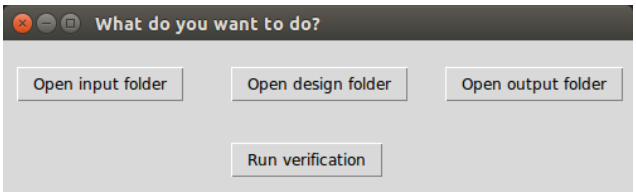

Figure 5: GUI controlling the tool.

The user can open the source folder using the first button to add all source files he wants to use, then he can do the same for every design that has to be tested by clicking the second button. Clicking the “run verification” button runs the tool and places all comparison files in the result folder. These comparison files are also waveJSON files that can be opened using WaveDrom.

IV. Technical Analysis

A. *Testbench creation*

1) *Testbench specification*

This section corresponds to the purple part of figure 3. To be able to create testbenches, we first have to define how they have to look. This depends on which framework is used in the simulation step (blue part of figure 3). In our case this is the VUnit framework, because it is one of the most advanced frameworks available. It’s also an open source framework, which makes it easy to work with. Other open source frameworks include: CoCoTB [3], SVunit [4], OSVVM [5] and UVVM [6].

VUnit testbenches are very similar to traditional testbenches, which means the learning curve is not very steep.

2) Source file

The waveJSON format was chosen because it was also the source format for WaveDrom, which is used to visualize wave traces and because it is an easily processable format. However, a standard WaveDrom file did not hold enough information to directly create a testbench from it. To solve this problem, some extra elements were added to it. This did not interfere with WaveDrom's ability to generate wave traces from the source file. The elements added to the standard waveJSON format include:

- The exact name of the unit under test
- The VHDL signal type for every signal
- The direction of the signal (input or output)

3) Conversion script

Now that both the source file and the testbench format are specified, the next step is to build a script that will convert one format to another. This is done using Python, because it is a flexible and easy to use programming language. It also has a large and active community, which means many examples are available online.

B. Wave trace analysis

This section corresponds to the green part of figure 3. After simulation a test will either pass or fail. This functionality is integrated in VUnit. The errors that are logged during the simulation are used to generate a new waveJSON output file that will show all the errors. The WaveDrom generated image from one of these files is shown in figure 4.

V. Example

In this section an example test is shown. An AND-gate design is tested for compliance with the specification

$$A \,\&\, B = F \tag{1}$$

The source file is shown below.

```
{"name": "andGate_timed", "test" : "andgate_full",
 "description": "test all possible inputs for an AND-gate",
 "signal": [
  ["CLK",
   {"name": "CLK", "wave": "p......", "type":"std_logic"}],
  ["IN",
   {"name": "A", "wave": "01010..", "type": "std_logic"},
   {"name": "B", "wave": "0..1010", "type": "std_logic"}],
  ["OUT",
   {"name": "F", "wave": "0..10..", "type": "std_logic"}]]
}
```

Figure 6: Source file for a full AND-gate test.

From this source file the documentation wave traces can be generated using WaveDrom. They are shown in figure 7.

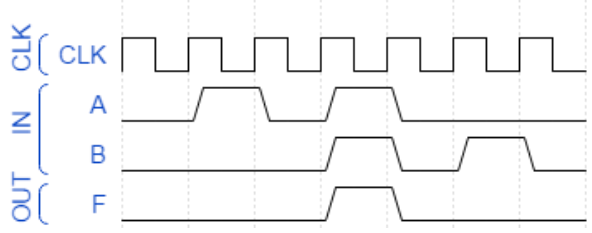

Figure 7: Documentation wave traces generated from the code in figure 6.

Then, a testbench is created from this same file. After simulation the logged errors are checked. In this case no errors were logged and the simulation succeeded. The AND-gate design complies with the specifications described in the source file. The result file will be equal to the source file, because the simulation wave traces are equal to the documentation wave traces. This file can be converted into an image by WaveDrom. This results in image 8.

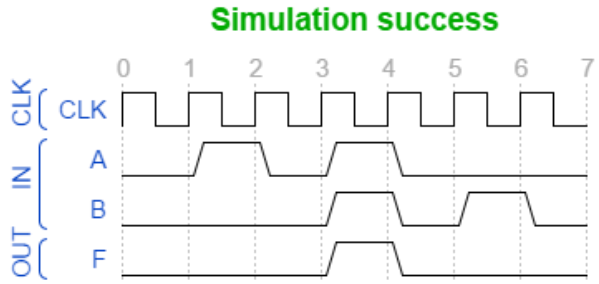

Figure 8: AND-gate test result.

VI. Benefits and Shortcomings

The tool has many benefits over the traditional validation system. For example, User input has been reduced to a single file: instead of the user needing to create a testbench, analyze the simulation wave traces and document the design, all this is now done automatically.

It does however also have some shortcomings. Some of the main problems are discussed here. First, because it has many dependencies and there is no installer available, it is not very easy to install. Installation is also only possible on a Linux device. The software was developed on Ubuntu 14.04 LTS. Secondly, the system is not suited for large tests, because either the source files or the resulting wave traces would become too large to clearly display. Then again, this is a problem other system also have to deal with. Large tests are tests where the amount of input and output ports of a design exceeds the maximum displayable amount or where the amount of clock cycles in the wave traces exceeds the maximum displayable amount.

VII. Conclusion

This section will try to provide an answer to the questions in chapter II.

A. Self-checking testbenches and documentation wave traces

As the source file is a waveJSON formatted file suited for use with WaveDrom, it is possible to directly generate wave traces from it. As the desired functionality is fully described by the source file, these wave traces can be used to document the design. This provides half of the functionality required to answer the first sub-question. The other half was made possible by adding some extra fields to the original WaveDrom file. By adding all the required information for creating self-checking testbenches. Using a conversion Python script and a template file it is possible to build fully fledged self-checking testbenches.

The resulting system does indeed fit the requirements. This means that it is indeed possible to design a system that can

create self-checking testbenches and documentation wave traces based on the same source.

B. Optimizing wave trace analysis

As the testbenches generated by the tool are self-checking, analyzing the tool becomes a lot easier. If a test succeeds, that means the design complies with the specification described in the source file. If a test fails however, it does not comply. In this case wave trace analysis comes down to comparing the simulation wave traces to the documentation wave traces. To do this, the tool creates a new waveJSON file where the differences between the two are described. This file can be visualized by WaveDrom.

C. Streamlining the process of validation

The tool facilitates the process of testbench creation and wave trace analysis, while at the same time minimizing user input and creating documentation wave traces to help document the design. Considering the limitations, we can say this tool indeed streamlines the process of validation up to a certain point. The software designed in this project can act as a proof of concept and lays the foundation for further improvement.

D. Benefits over existing verification frameworks

As the tool is unique in its kind, it does in its way provide benefits over existing frameworks. It extends VUnit and adds extra functionality. It does this by concentrating user input into one file and simplifying wave trace analysis.

It is important however to consider that this is only the first version of this system, which still has many flaws. While it extends VUnit, it also takes away a lot of the functionality the VUnit framework offers. By better supporting this framework an even better system could be made. Also, it could be an improvement to support other frameworks, like CoCoTB, as well.

REFERENCES

[1] A. Chapyzhenka, "WaveJSON · drom/wavedrom Wiki," WaveDrom, 5 May 2014. [Online]. Available: https://github.com/drom/wavedrom/wiki/WaveJSON. [Accessed 13 Oktober 2015].

[2] A. Chapyzhenka, "WaveDrom - Digital timing diagram everywhere," WaveDrom, 2011. [Online]. Available: http://wavedrom.com/. [Accessed 30 September 2015].

[3] PotentialVentures, "Welcome to Cocotb's documentation!," PotentialVentures, 2014. [Online]. Available: http://cocotb.readthedocs.io/. [Accessed 14 04 2016].

[4] agilesoc, "SVUnit," agilesoc, 5 June 2015. [Online]. Available: http://www.agilesoc.com/open-source-projects/svunit/. [Accessed 17 May 2016].

[5] OSVVM organisation, "Open Source VHDL Verification Methodology," OSVVM oranisation, 2013. [Online]. Available: http://osvvm.org/. [Accessed 2016].

[6] Bitvis, "UVVM - Overview," Bitvis, 2013. [Online]. Available: http://bitvis.no/products/uvvm/. [Accessed 2016].

[7] L. Asplund, "VUnit documentation," 2014. [Online]. Available: vunit.github.io/documentation. [Accessed 2016].

Contents

Chapter 1

Introduction

In software and hardware development, verification and validation is the process of checking whether a system meets specifications and that it fulfills its intended purpose. It may also be referred to as quality control. It is normally the responsibility of testers as part of the development lifecycle [1].
In other words, the questions

- Are we building the product right? (Verification)
- Are we building the right product? (Validation)

are becoming more and more important to designers as consumers become increasingly demanding.

The software development industry has invested enormously in new verification methods and is still building toward better verification. Software developers have an abundance of support tools available for their respective languages (e.g. C, Java, Python, etc.). These tools help the programmer by automating a lot of the actions required to write, validate, and document the code.

Hardware development has always lagged behind however. Support tools for hardware development languages (HDLs) like VHDL and Verilog are limited and often outdated. Modernising verification methods for HDLs is progressing more slowly than for programming languages. There is still a lot of room for improvement in this area.

In light of this, this report introduces a way to improve VHDL design verification. More specifically, provide a way to easily design tests and simulate them, while at

the same time helping to document the design and automatically comparing the simulation result to the design documentation (or specification). The operation of the resulting software is illustrated below.

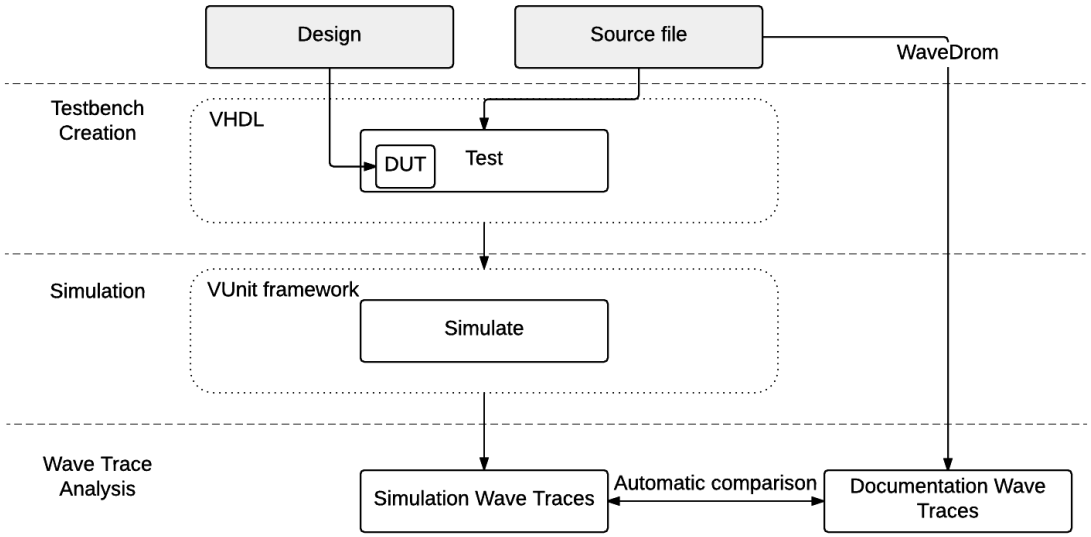

Figure 1.1: Operation overview of the new system

The tool starts from two input files: the design to be tested and the source file, which specifies a test. A test checks the design for compliance with its specification. These two files are processed and a VHDL test is built for the design. This test is simulated and its results are saved as the simulation wave traces. These wave traces show how the design behaves by visualising the input and output signals during the test. At the same time the documentation wave traces are derived from the source file. These wave traces show how the system should behave. They are compared to those generated by the simulation. If the simulation result is as expected, the design has passed the test. It is compliant to what is specified in the documentation.

To understand the purpose and use of the proposed tool, a short introduction in the verification process is necessary.

Whenever a hardware component is implemented, its behavior is defined in the design file. The behavior of the component is then often documented visually by showing input and expected output wave traces. Wave traces are the visualisation of the input and output signal values over time. An example for an AND-gate is shown below. Input signals (A & B) are connected to the design's input and the output signal (F) represents the expected output of the design.

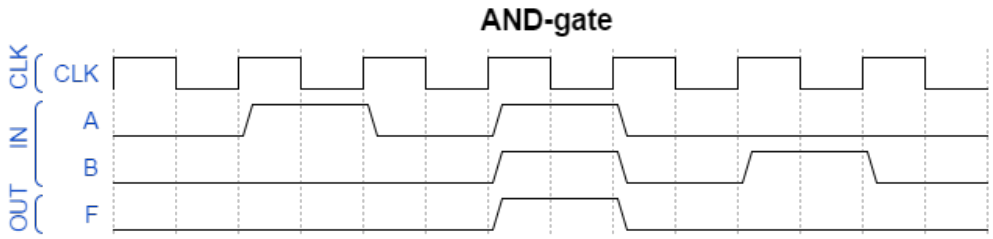

Figure 1.2: AND-gate wave trace example

The behavior is tested by attaching it to a so called testbench. Historically, this testbench is also written in a HDL. It describes the environment in which the design is to be tested. In this testbench one or more tests can be performed by attaching certain input signals to the designs input ports. Creating this testbench is the first step in the verification process. The design being tested is often referred to as the Device Under Test (DUT) or the Unit Under Test (UUT). Figure 1.3 illustrates the hierarchy of a testbench.

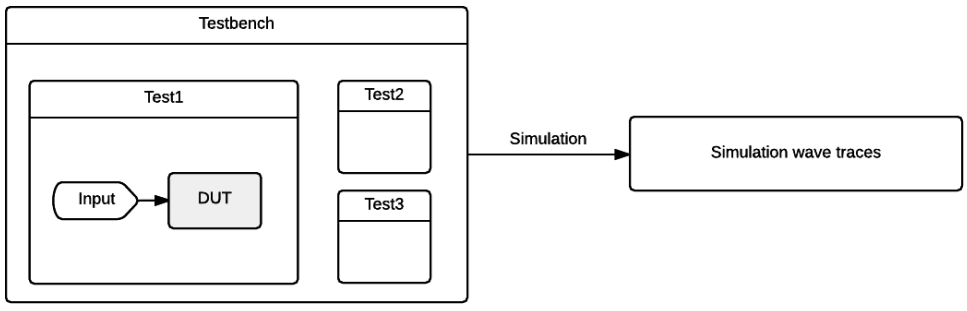

Figure 1.3: Hierarchy of a testbench

The second step of the verification process is to simulate the tests described in a testbench. A simulation yields output signals depending on the input signals and the behavior of the DUT. Figure 1.4 shows an example of a simulation output (simulation wave traces). If a design functions correctly, these wave traces are equal to those in the documentation.

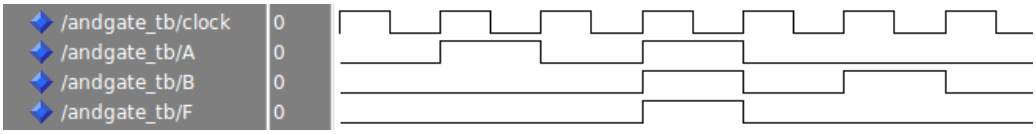

Figure 1.4: Modelsim simulated wave traces

To verify a design, ideally, every possible input has to be simulated in a test and have its corresponding simulated output compared to the expected output specified in the documentation. This would mean that the tests cover 100% of the design code. In practice trying to reach this percentage is not economically desirable, but designers still strive to reach an acceptably high coverage percentage. Checking whether test results are in accordance with the documentation is the last step in the verification cycle.

The complete verification cycle is shown in figure 1.5. Starting from a VHDL design, a) a testbench is created, b) then it is simulated and c) finally its wave traces are analysed.

Each step can be optimized in its own way. The proposed tool will focus on streamlining the first and the last step, relying on existing simulation tools to handle the second step. It will use a new source file to help generate documentation wave traces, while at the same time using that file to create testbenches and analyse simulation wave traces. The improved verification cycle is shown in figure 1.6. Figure 1.1 shows these steps in detail.

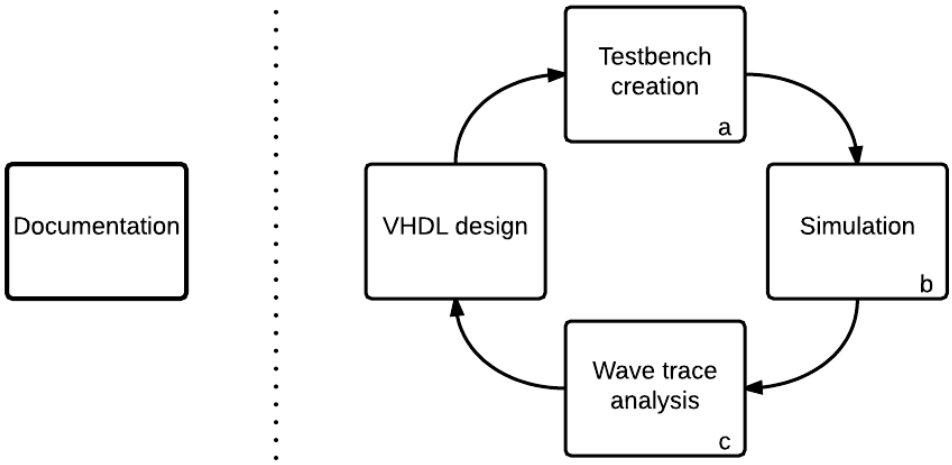

Figure 1.5: Traditional verification cycle

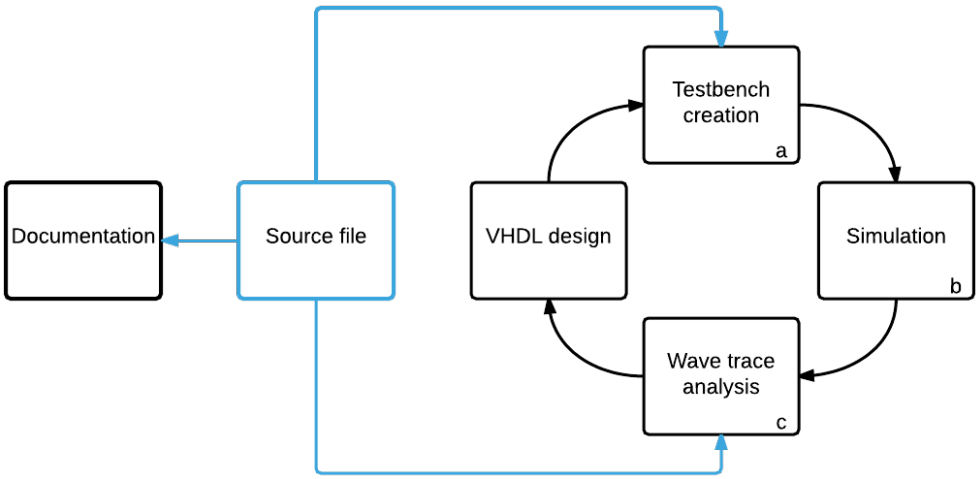

Figure 1.6: Improved verification cycle

Chapter 2

Problem analysis

The classic (V)HDL verification process is shown in the figure 1.5. Starting from a VHDL design a testbench is created. The testbench is then simulated, which yields the simulation wave traces. These wave traces are then checked for compliance with the documentation wave traces. If the design behaves as the documentation specifies, the verification process is complete. If the design does not behave accordingly, the design is reviewed and the process is run again. The behavior of the design is described in the documentation.

The least efficiënt steps in this process are the creation of the test benches (top) and the visual comparison of wave traces (bottom). They will be the primary elements of focus in this report.

The first element of focus is the creation of testbenches. Designers either have to rely on older, outdated, testbenches who might not be self checking or design a new testbench manually. Self checking testbenches are testbenches that report any irregular behavior automatically, whereas conventional testbenches simply simulate output signals and leave the analysis to the designer. Testbenches are often at least partly written manually and creating them is a repetitive task. This means writing a test for every aspect of a design is time consuming and prone to human error. If this process could be automated, a lot of time and effort could be saved. At the same time writing the documentation is a tedious task, where every aspect has to be detailed. Documentation often includes example wave traces. As can be see from image 1.2 and 1.4, which are documentation wave traces and simulated wave traces of a certain feature respectively, these wave traces are exactly the same (considering the design functions as documented). Although the way they were created differs

greatly, the documentation and test result hold the same information. In theory it would be possible to devise a system that could build both the documentation wave traces and the test for a specific feature from the same information. This eliminates the need to create testbenches manually, while still offering the same functionality.

In short, the system should be able to automatically build a self checking testbench as well as the corresponding wave traces based on the same information.

The second element of focus is comparing the expected output in the documentation to the simulated output of the testbenches, or simply wave trace analysis. In other words, making sure the design functions according to the documentation. Self checking testbenches perform an analysis during simulation and show errors to the user by logging them to a file or terminal, but testbenches that are not self checking leave all wave trace analysis to the user. Comparing wave traces visually is not ideal, because the complexity of some tests and their corresponding wave traces is considerable. This often results in a staring contest between designer and simulation result. Because of this, the proposed solution will not only build self checking testbenches, but will process the analysis result and show it in a user friendly and easily understandable format.

In short, the system should be able to take the error reports logged by the self checking testbench and show them in an easily understandable way to a user.

The resulting software package should be easy to use, that is why a simple GUI will be added where the user can input the source file and the VHDL design to be tested. From this information the testbench can be created and simulated by simply pressing a button. Finally the simulated wave traces are automatically compared to the documentations and the result is shown.

Lastly a short analysis will be made regarding the use of this software and the benefits it provides over existing frameworks.

To summarise, this report will try to answer the following questions:

- Can we design a system that will streamline the process of design validation?
  - Can we design a system that can create self checking testbenches and documentation wave traces from the same source?
  - Can we optimise wave trace analysis based on this system?
- Does this system provide any benefits over existing verification frameworks?

Chapter 3

Functional analysis

The goal of this project is to create a tool that can streamline the testbench creation and the wave trace analysis processes to help developers validate their design, while at the same time generating wave trace images to aid in documenting a design. The original validation process flow can be seen in figure 1.5. To improve this process a new source file is introduced. Figure 1.6 shows the improved information flow of the validation process. Where the original system required (more) user input at both the documentation side and the validation side, this system bundles most user input into one file, reducing the total amount of user input and saving a considerable amount of time. An overview of how it works is shown in the figure below.

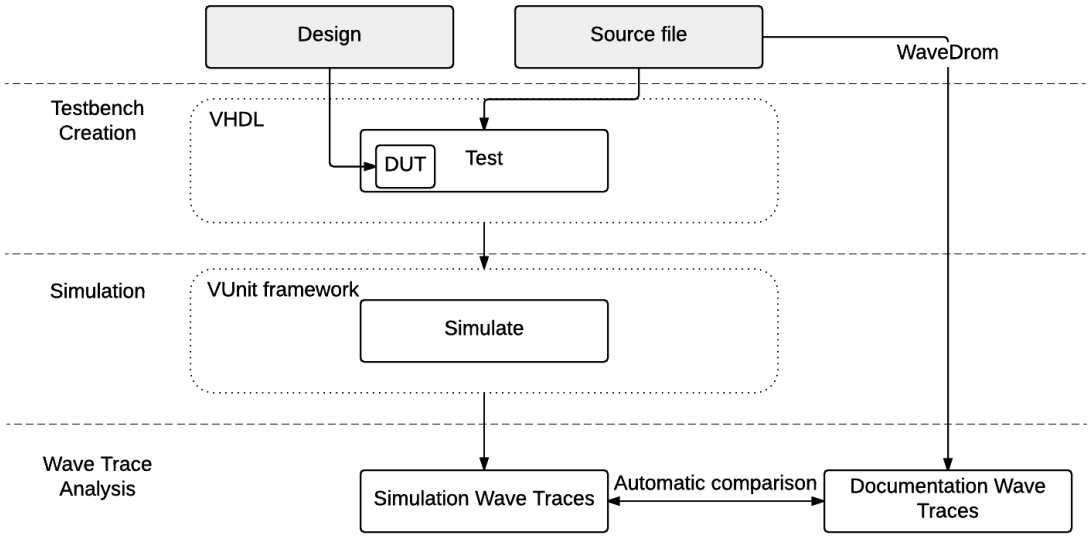

Figure 3.1: Operation overview of the new system

The tool will complete a full cycle of the validation process. It starts from the user input files: the source file and the design file. And then continues in three steps:

First, the tool can create testbenches automatically and generate documentation information using a source file. This source file is a waveJSON [9] file based on standard WaveDrom [2] files. WaveDrom is an open source engine for generating wave traces from waveJSON files. The tool can create a testbench for every source file provided in mere seconds, faster than a developer could create just one testbench manually. These testbenches are designed according to the specifications of a VUnit [3] testbench. VUnit is a framework that extends the VHDL language.

Then, using VUnit provided functionality, the created testbenches are simulated by an external simulator. This can be any simulator supported by the VUnit framework.

Finally, instead of the manual comparison of two wave traces, it offers a way to quickly find non corresponding wave traces in a visual way. It returns a waveJSON file similar to the source file, which can be visualised using WaveDrom. Figure 3.2 illustrates this concept. This visual representation will be accompanied by an error message specifying the error.

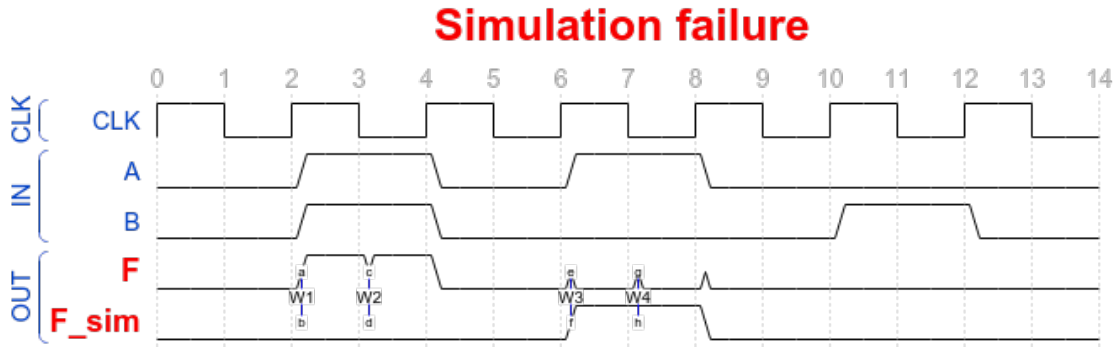

Figure 3.2: Simulation failure example

W1: Expected sig_F = '0', got sig_F = '1' at n = 2.

W2: Expected sig_F = '0', got sig_F = '1' at n = 3.

W3: Expected sig_F = '1', got sig_F = '0' at n = 6.

W4: Expected sig_F = '1', got sig_F = '0' at n = 7.

Log message 1: Example logged error messages

The two features are bundled together in one tool controlled by a simple GUI.

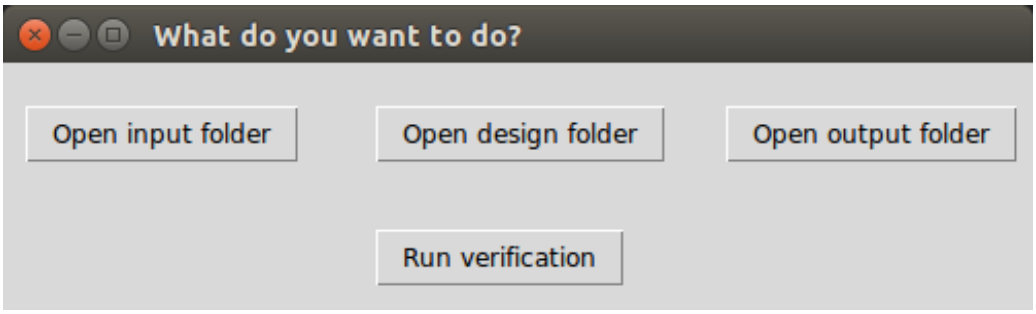

Figure 3.3: Look of the user interface

The GUI allows the user to open the input folder where WaveDrom source files should be added, the design folder where the design to be tested should be added and the output folder, where the resulting files will be stored. The "run verification" button creates a testbench for every source file in the input folder, simulates the tests, verifies them and stores the results in the output folder.

To summarise, the tool will convert

- Source files for every feature to test
- The design under test VHDL description

into

- A VHDL test for every source file
- waveJSON wave trace analysis files for every test

There are many benefits to this system. The user can be sure the test that has been generated does not hold any mistakes (at least if we assume no mistake has been made in creating the source file) as it is directly derived from the specifications of the device. Also, validating a device becomes easier compared to what it used to be - (create testbench,) simulate, analyse wave traces, improve design, repeat. One click offers an immediate validation. The test will either pass or fail. In the latter case extra information is immediately available.

It is however not a full validation system on itself. It requires the use of existing frameworks to function. It offers an easy to use and approachable way to use these frameworks while at the same time adding functionality to them. In this case the VUnit framework was chosen, but it could also be adapted to run on another framework like CoCoTB [4]. It is a useful extension of these frameworks for all those who want to validate a design and document its functionality.

Chapter 4

Technical analysis

This chapter will analyse each part of the system separately based on the system overview in figure 4.1.

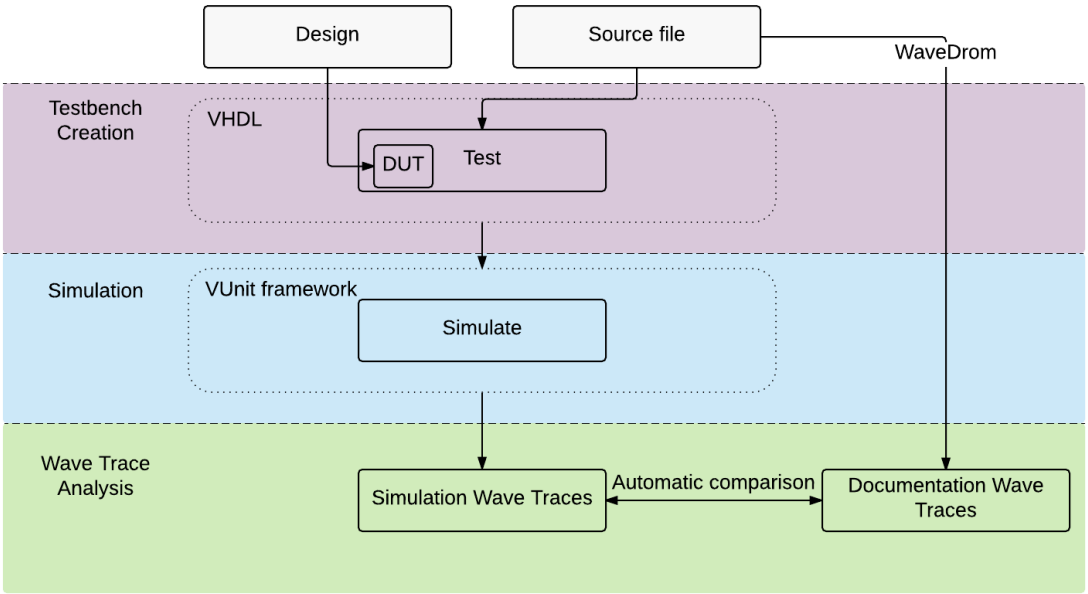

Figure 4.1: Color coded overview of the program operation

4.1 Testbench creation

Creating testbenches requires several steps. First the specifications for the testbench have to be determined, then, the way they will be created has to be determined and finally the source material (i.e. what the testbench will be created from) has to be chosen. Creating testbenches corresponds to the purple part in image 4.1.

4.1.1 Testbench specifications

It is important to define the way testbenches should be constructed. Before it is possible to do this, however, the specifications for the testbenches have to be determined. This depends on the framework used in the simulation step (blue part). Several testbench frameworks are available that improve on the standard VHDL testbenches. They offer features such as unit testing, (conditional) logging packages and many more. Some even offer the possibility to write testbenches in languages other than VHDL.

Some of the most known frameworks are listed below. The language they extend and their most important features are mentioned. These frameworks are all open source.

VUnit : VHDL, (System)Verilog

- Unit testing
- Python test runner
- Advanced logging libraries
- Multi simulator support

CoCoTB : VHDL, (System)Verilog

- Python testbenches
- Python test runner
- Multi simulator support

SVUnit [5]: (System)Verilog

- Unit testing

OSVVM [6]: VHDL

- Maximum test coverage support
- Randomised testing

UVVM [7]: VHDL

- Advanced logging libraries
- Reusable and readable tests

The frameworks with the most advanced features and support for both (System)Verilog and VHDL are VUnit and CoCoTB. The framework suggested by the promoters of this project is VUnit. It will be the framework supporting this system.

The VUnit framework aims to bring the best software testing practises to HDL languages. A big part of that is supporting unit testing, hence the name V(HDL)Unit. The VUnit documentation can be found in the vunit documentation [8]. VUnit testbenches do not differ a lot from regular testbenches. The only difference is that all test have to be written inside the same process as seen in the example below.

```
main : process
begin
test_runner_setup(runner, runner_cfg);
while test_suite loop
  if run("test_pass") then
    report "This will pass";
  elsif run("test_fail") then
    assert false report "It fails";
  end if;
end loop;
```

Code 4.1: Main test loop for a VUnit testbench

Multiple unit tests can be implemented using the if-elsif mechanism. All tests are run by the Python test runner, which can be configured according to the user's wishes. This means that a unit test can be simulated by a Python command. These tests can be run separately thanks to the VUnit framework. The framework offers the possibility to set the simulator backend that will simulate all the tests. How to set up the VUnit framework and run tests can be found in the VUnit documentation.

4.1.2 Conversion script

With the form of the testbenches defined, a general construction method can be implemented. This is done using the popular programming language Python, because of its clarity, flexibility and ease of use. On top of this it is the language used to build the VUnit framework, which makes it easier to integrate VUnit into the software.

The conversion script will read the wave trace file, convert it to a VUnit compatible testbench and run it using the VUnit framework.

A lot of information relevant to creating testbenches could be extracted from the device description, such as the name, type, and direction of all ports. For now, however, the conversion script expects all information to be available in the wave trace files. An optimization can be made at a later time.

4.1.3 Source file

The source file is the source for creating the testbench, it contains all the information necessary to create a testbench, while still being readable and understandable. Information should be easily extractable and processable. These constraints are all met by WaveDrom Files.

WaveDrom is an open source timing diagram rendering engine that converts waveJSON input files into the corresponding timing diagram. These waveJSON files are considered the source file, but will also be referred to as WaveDrom (input) files in this report.

WaveJSON [9] is an application of the JSON format. The purpose of it is to provide a compact exchange format for digital timing diagrams.

Standard WaveDrom files do not hold enough information to create a testbench. However, it is possible to add new JSON fields to these WaveDrom files without compromising the generated wave traces. This way any information needed can be integrated into these files.

4.2 Wave trace analysis

This part corresponds to the green part of image 4.1. After simulation a test will either pass or fail. This functionality is integrated in VUnit. During simulation the output is compared to the expected output and whenever they are not equal, the test will automatically fail and an error message will be generated. This error message is logged to a CSV file using the conditional logging functionality of the VUnit Check library.

The logged errors can then be processed by a script to create waveJSON files in which errors are clearly marked. This script will be written in Python and will create WaveDrom files based on the original input WaveDrom file.

Chapter 5

Technical design

5.1 Testbench creation

This chapter holds a step by step description of the development of the software. Every new element is introduced by means of increasingly complicated example source files.

5.1.1 Simple example

The WaveDrom file

Although the primary goal of this product is to create testbenches for timed designs, a basic first draft of the tool's conversion script can be created based on a simple combinatorial design.

The example this draft was based on was an AND-gate design. The AND-gate design can be found in appendix C.1. All designs used in this project can be found online [10] in each example's respective design directory (vhdl_files/src).

A WaveDrom file for describing the wave traces of a purely combinatorial AND-gate is shown in code 5.1.

```
1 {"signal": [
2   {"name":"A", "wave":"0"},
3   {"name":"B", "wave":"0"},
4   {"name":"F", "wave":"0"}]
5 }
```

Code 5.1: Standard WaveDrom description for a combinatorial AND-gate

Where A and B are considered input signals and F is the output signal. This yields the following wave trace image, which shows that F should be low when both inputs are also low.

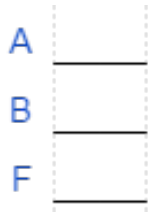

Figure 5.1: Wave traces corresponding to description in code 5.1

This is neither very clear, nor does it hold enough information on its own to create a testbench. To be able to create a testbench several more fields containing extra information should be added.

The necessities for creating a testbench are:

- The exact name of the unit under test
- The VHDL signal type for every signal
- The direction of the signal (input or output)

These fields have been added to the WaveDrom file together with a name and description for this test. The new WaveDrom file is shown in code 5.2:

```
{"name": "andGate", "test" : "andgate00",
"description": "Input A = '0' & B = '0' should produce a low
    output",
"signal": [
  ["IN",
    {"name": "A", "wave": "0", "type": "std_logic"},
    {"name": "B", "wave": "0", "type": "std_logic"}],
  ["OUT",
    {"name": "F", "wave": "0", "type": "std_logic"}]
]}
```

Code 5.2: Extended WaveDrom description for a combinatorial AND-gate

Where "name" should be the exact name of the unit under test and the signal names have to be exactly the same as the port names in the uut design. Each port should be defined below their corresponding label of direction. All input files must be defined below the "IN" label and all output signals under the "OUT" label. Now all information needed to create a testbench is available in the source file.

Creating VHDL code from the WaveDrom file

The information in this file has to be translated to VHDL code. More specifically the information in the WaveDrom file should be converted into:

- Signal declarations for every signal
- DUT declaration
- DUT port map
- Input signal stimulus
- Output signal checking

These are the elementary parts of a VHDL testbench. The python JSON package enables an easy conversion of information by reading all of the waveJSON information and placing it in a Python JSON container. This way, data in the Python code has the same structure as the data in the WaveDrom file. Because all the information is now available in the Python program all that has to be done is manipulate standard strings to fit the current design.

For example the string manipulation code for signal declarations is given below.

```
def generate_signal_declaration(json_signal):
  signal_name = json_signal["name"]
  signal_type = json_signal["type"]

  #example: "signal sig_example : std_logic :='0';
  return "signal sig_" + signal_name + " : " + \
          signal_type + " := '0';"
```

Code 5.3: Generating signal declarations in Python

This method reads the information regarding a single JSON signal and produces a VHDL signal declaration string which initializes the signal to '0'.

In the same way signal stimuli and the UUT port map can be generated.

```
def create_stimulus(input_signals):
  stimulus = ""
  for signal in input_signals:
    if len(signal) > 0:
      value = signal["wave"]

      #example: "sig_example <= '1';"
      stimulus += "sig_" + signal["name"] + " <= '" + value + "
    ';\n"
  return stimulus
```

Code 5.4: Generating signal stimuli in Python

This method needs a list of all input signals in JSON format. It creates a VHDL assignment assigning them the value contained in the "wave" field and returns that string. The result can be directly added to the testbench.

```
def generate_port_map(json_signal_set):
    port_map = "PORT MAP(\n"
    first = True
    for signal_type in json_signal_set:      # IN and OUT.
      if len(signal_type) > 0:              # JSON allows empty
     elements
        for signal in signal_type:
          if len(signal) > 0:          # JSON allows empty elements
            if first:
              #example: "example => sig_example"
              new_map = signal["name"] + " => sig_" + signal["name"
     ]
              first = False
            else:
              new_map = ",\n" + signal["name"] + " => sig_" +
     signal["name"]
            port_map += new_map
    port_map += " );"
    return port_map
```

Code 5.5: Generating a port map in Python

The port map is generated from a list of all signals. As in the input WaveDrom file this list is composed of two sublists: the signals defined under "IN" and those under "OUT". These lists are iterated through and for each element that is not empty a new element is added to the port map. The final port map is returned and is ready to be added to the testbench file.

Converting the WaveDrom file to a VHDL testbench

Every testbench is structurally the same at its base. The main elements are the signal declarations, the device under test (DUT) declaration, the port mapping of the DUT, the clock driving process and the stimulus process. As the previous paragraph explains, these elements can all be generated from the WaveDrom file. All that is left to do is organize these elements in the proper structure. To do this a template is used. This template defines the structure of the testbench by placing keywords that will be recognized by the program. when creating a test, the tool will replace these keywords with design specific code. In this template the VUnit checking capabilities are integrated, allowing for conditional logging. The final version of the template can be found in appendix D.

The space for input and output signals is marked by the keywords '–# Input Signals' and '–# Output Signals' respectively. Every signal declaration is created and placed below the corresponding keyword.

Using this method the first fully functional testbench was created that can be run using VUnit. The creation is however limited to combinatorial designs.

5.1.2 Clocked designs

The next step is to move on to Clocked (or timed) designs, which will be the main focus of this project. The design used to test this step does not differ from the previous AND-gate design except that the gate will be triggered by a clock signal. This allows the creation of more extensive tests. The AND-gate VHDL design can be found in appendix C.2.

The WaveDrom source file for testing all possible inputs of an AND-gate is shown in code 5.6. The generated wave traces are shown in fig 5.2.

```
1  {"name": "andGate_timed", "test" : "andgate_full",
2  "description": "test all possible inputs for an AND-gate",
3  "signal": [
4    ["CLK",
5      {"name": "CLK", "wave": "p......", "type":"std_logic", "
       period":"2", "clock_period": "20"}],
6    ["IN",
7      {"name": "A", "wave": "01010..", "type": "std_logic", "period
       ":"2"},
8      {"name": "B", "wave": "0.....1.0.1.0.", "type": "std_logic"}]
       ,
9    ["OUT",
10     {"name": "F", "wave": "0.....1.0.....", "type": "std_logic"}]
11 ]}
```

Code 5.6: JSON source file for a full AND-gate test

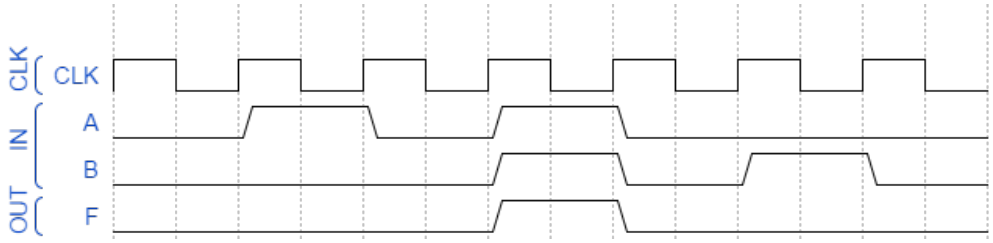

Figure 5.2: Wave traces generated by WaveDrom based on the source file in code 5.6

The clock signal is distinguished from the other input signals by adding a new label.

Firstly, until now signal driving and checking was quite straightforward. The signals were given the value specified in the WaveDrom file and after a short wait the output was checked. In timed designs, however, ports can have different values every time the clock value changes. Because of this it is necessary to loop over the values of the signals and load a new value every clock cycle.

Secondly, the new clock signal introduces the need to implement a clock driving process and to define a clock period.

Driving and checking the signals

The value of every input signal can change every clock cycle. This means that a new value should be loaded every clock cycle. To be able to do this inside a VHDL testbench two things are needed. First, all future values have to be known on simulation start and secondly, the current clock cycle has to be known at all times. For every signal an array will be defined that holds all values that will ever be assigned to it. The index of every element in that array corresponds to the time step at which it will be assigned, which is why the current clock cycle must always be known.

Creating these arrays requires three things. A conversion of the JSON signal value representation to a VHDL representation, knowing the logic type of the signal concerned, and knowing the length of the array.

Converting the JSON signal value boils down to expanding the "value" string and converting the separate characters (e.g. the "wave" field of a waveJSON signal of "0.1." will be converted into ('0', '0', '1', '1')). A '.' represents the repetition of a previous value.

To create an array, the logic type it holds must also be known, because an array type can only be defined if the type contained in it is known. For example, the array type for an array of length clock_cycles containing std_logic elements is defined as seen in code 5.7.

```
type std_logic_array is array (0 to clock_cycles - 1) of
    std_logic;
```

Code 5.7: Declaration of an std_logic array type of lenght clock_cycles

This array type must be defined before an instance of it can be created. Consider the JSON input signal "sig_example" of which the signal wave is defined as "01" that has logic type "std_logic". If no type exists that can hold std_logic elements this array type is first defined as in code 5.7. After this the actual array can be declared. The array holding the values will be named "sig_example_values" and its definition will look like in code 5.8:

```
constant sig_example_values : std_logic_array := ('0', '1');
```

Code 5.8: Definition of a std_logic array

We assume that the test is defined for two clock cycles and the clock_cycle constant is equal to 2. This constant is derived from the amount of values in the "wave" field of a waveJSON signal and should be the same for every signal.

These arrays are not only used to store the future values for all input signals, but are also used to store expected values for output signals. Every time the input changes, the output can change too. Because the current clock cycle is known the simulated output can be compared to what is expected at this point in the simulation. This comparison is what is called a signal check and the result of the test will depend on whether the two values are equal or not.

Now all that is left to do is to assign the next value and compare the output signals every clock cycle. This is illustrated in code 5.10.

Clock driving

Defining a clock driving process and clock period inside a testbench does not require a lot of effort. A new keyword was added to the testbench for both. The keywords '–# Constants' and '–# Clock Driver" respectively are placeholders for the constant declaration and clock driving process respectively.

```
constant internal_clock_period : time := 20.0 ns;
```

Code 5.9: Definition of a std_logic array

```
Clk_proces : process
begin
  if (EndOfSimulation = '0') then
    internal_clock <= '1';
    wait for internal_clock_period / 2;
    internal_clock <= '0';
    wait for internal_clock - internal_clock_period / 2;
  end if;
end process;
```

Code 5.10: Signal driving in clocked designs

The clock driving process is not design dependent and is simply added to the test-bench whenever a clock signal is detected in the WaveDrom file. The clock period can be specified (in ns) in the WaveDrom file as seen in code 5.6, but will default to 20 ns if that is not the case.

Because it is unknown whether the actual clock signal will be high-to-low or low-to-high, an internal clock signal is introduced which will trigger the driving process. Every input signal, including the actual clock, will be driven on the rising edge of the internal clock and every output signal will be checked on the falling edge. The internal clock is always the same, but the actual clock can be different for every design. Figure 5.3 shows how the internal clock relates to some possible internal clocks.

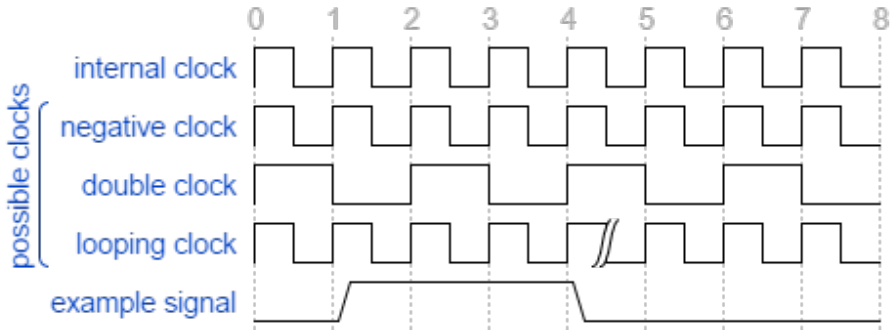

Figure 5.3: Examples of actual clock signals compared to the internal clock

Actual clock signals can change twice every internal clock period (e.g. negative clock and looping clock) and have to be driven on the falling edge as well as the rising edge of the internal clock. For the timed AND-gate example the driving and checking process of the signals is shown in code 5.11.

```
while (n <= clock_cycles -1) loop
  wait until rising_edge(internal_clock);
  sig_CLK <= sig_CLK_values(2*n);
  sig_A <= sig_A_values(n);
  sig_B <= sig_B_values(n);

  wait until falling_edge(internal_clock);
  -- CLK has twice as many values as other signals
  sig_CLK <= sig_CLK_values(2*n_1);
```

Code 5.11: Signal driving in the AND-gate test derived from the source file in code 5.6

This means that the amount of clock cycles should be known within the testbench. The conversion software calculates the amount of clock cycles in the WaveDrom file and adds a clock_cycles constant to the testbench using another keyword.

This internal clock has another advantage. WaveDrom allows for signals to have a relative period. The relative period can be set for every signal separately using the "period" field. A relative period of 2 means that a signal will keep every value denoted in the "wave" field for 2 time steps. This way, for example, the actual clock can run twice as slow as the internal clock. This is illustrated in figure 5.3 by "double clock". This allows output signals to change on the falling edge of a high-to-low clock signal (e.g. "example signal" in relevance to "double clock"), which would not be possible if all signals were driven on the first edge of the actual clock signal. The internal clock period will define the length of a time step and will rise and fall within one time step. This way each signal is driven or checked every time step regardless of their relative period.

5.2 Vectors and repetitions

At this point it is possible to create testbenches for timed designs. To test the robustness a more complicated design is tested: the UART design example available in the VUnit documentation [8]. The UART design can be found in appendix C.3.

5.2.1 Vectors

In a first test the UART will be required to send the content of the parallel input 'data' (1 byte) over its serial output 'tx'. The scope of this first test will be limited to the first 8 clock periods, right before the actual sending of the data. The WaveDrom input code and corresponding wave traces are shown below.

```
1  {"name": "uart_tx", "test" : "uart_send_1_byte_no_wait",
2  "description": "Send one byte with a parallel to serial uart (
       actual sending excluded)",
3  "signal": [
4    ["CLK",
5      {"name": "clk", "wave": "n.......", "type":"std_logic", "
       period": "2", "clock_period": "20"}],
6    ["IN",
7      {"name": "tvalid", "wave": "0.1..0.........", "type": "
       std_logic"},
8      {"name" : "tdata", "wave": "=.=..=.........", "data": ["0",
       "249", "0"], "type" : "std_logic_vector", "vector_size" : "8"}
       ],
9    ["OUT",
10     {"name": "tx", "wave": "1....0.........", "type": "std_logic
       "},
11     {"name": "tready", "wave": "01.0...........", "type": "
       std_logic"}]
12 ]}
```

Code 5.12: Source file for creating the first transmission test for the UART design in C.3

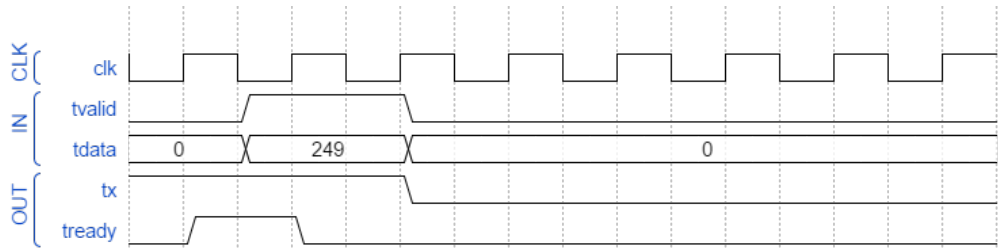

Figure 5.4: Wave traces generated by WaveDrom from code 5.12

The image above also illustrates the benefit of the relative period set in the "clk" signal. Where each clock cycle would take only one internal clock cycle to complete if the "period" field were not, it will now take two. This allows the "tvalid" signal to transit from high to low halfway through a clock cycle. The "clock_period" field will make sure every actual clock cycle will still only take 20 ns, instead of the standard 20 ns per internal clock cycle, which would total to 40 ns per actual clock cycle.

This new design poses a new challenge , however. This is the first design that uses vectors as a port type. WaveDrom supports the use of vectors with the help of the '=' character and the 'data' field. The approach to converting the WaveDrom vector information to VHDL vector information is not very different from the approach regarding single bit ports. The only difference lies in the newly added "data" and "vector_size" fields. The information contained in the "data" field has to be linked to the correct '=' character before converting the signal to VHDL code. This last step requires some extra information, however. In VHDL it is necessary to know the size of a vector before it can be declared. This information has to be added to the wavedrom file. The field 'vector_size' holds this information.

Time loops

The second test for the UART design is an extension of the first. This time the design will be allowed to send the data over its 'tx' output.

```
{"name": "uart_tx", "test" : "uart_send_1_byte",
"description": "Send one byte with a parallel to serial uart.",
"signal": [
  ["CLK",
    {"name": "clk", "wave": "n......|", "type":"std_logic", "
    period": "2", "clock_period": "20", "loop_times" : ["10*434"]}
    ],
  ["IN",
    {"name": "tvalid", "wave": "0.1..0.........", "type": "
    std_logic"},
    {"name" : "tdata", "wave": "=.=..=........x.", "data": ["0",
    "249", "0"], "type" : "std_logic_vector", "vector_size" : "8"}
    ],
  ["OUT",
    {"name": "tx", "wave": "1....0........x.", "type": "std_logic
    "},
    {"name": "tready", "wave": "01.0.........x.", "type": "
    std_logic"}]
]}
```

Code 5.13: Source file for creating a second transmission test for the UART design in C.3

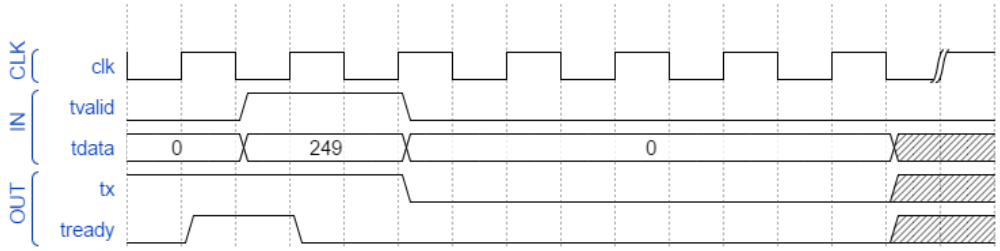

Figure 5.5: Wave traces generated by WaveDrom from code 5.13

Considering the design is created to send 10 bits (one start bit, 8 data bits and one stop bit) over its output line and every bit is held for 434 clock cycles, it would take a WaveDrom file of at least

$$8 + 10 * 434 = 4348 \tag{5.1}$$

clock cycles to create this test. This would create immense WaveDrom files and would nullify the benefits of this whole approach. It is vital to create a simple and clear way to implement this test in WaveDrom. To do this, two new WaveDrom characters are used.

The 'x' character is part of the WaveDrom character set and signifies an unknown value. When this character is read by the conversion software it is processed just like any other character, except when it comes to checking output values. In the testbench, if an 'x' is encountered for a certain signal the normal check is simply skipped. In other words, an 'x' in the WaveDrom file means that this signal will not be checked at this point. Because this character only affects the checking of signals, it has to be used in output signals and can not be used elsewhere.

The '|' character is also part of the WaveDrom character set and is used to indicate that for an undefined time a signal is not shown. The meaning of this character changes a little for this application. Here it can only be placed in clock signal and signifies that the simulation loops for a certain amount of clock periods. During this loop, the simulator will check whether the output signals remain unchanged. The amount of clock periods a simulation should loop has to be specified in the new loop_times field as shown in the example above. Multiple loops can be added by adding new '|' character and corresponding loop times, but, other than in an ordinary WaveDrom file, the wave value ('p' or 'n') has to be repeated after the loop character as seen in the next example.

Adding these two characters makes it possible to write the 4348 clock cycles long test in just 9, by adding a loop for $10 * 434 = 4340$ cycles. The downside, however, is that during this period the output can not change or is simply not checked.

5.2.2 Application example

The following test shows the use of these features. In this example a designer wants to check whether the device can send two consecutive bytes, but is not interested in the sending itself, because that has been tested in another test.

```
1  {"name": "uart_tx", "test" : "uart_send_2_bytes",
2  "description": "Send two consecutive byte with a parallel to
       serial uart.",
3  "signal": [
4    ["CLK",
5      {"name": "clk", "wave": "n......|n......|", "type":"std_logic
       ", "period": "2", "clock_period": "10", "loop_times" : ["10*43
       4", "10*434"]}],
6    ["IN",
7      {"name": "tvalid", "wave": "0.1..0...........1..0........."
       , "type": "std_logic"},
8      {"name" : "tdata", "wave": "=.=..=........x.==..=........x."
       , "data": ["0", "249", "0", "0", "127", "0"], "type" : "
       std_logic_vector", "vector_size" : "8"}],
9    ["OUT",
10     {"name": "tx", "wave": "1....0........x.1....0........x.", "
       type": "std_logic"},
11     {"name": "tready", "wave": "01.0.........x.1..0.........x."
       , "type": "std_logic"}]
12 ]}
```

Code 5.14: Source file for creating third transmission test for the UART design in C.3

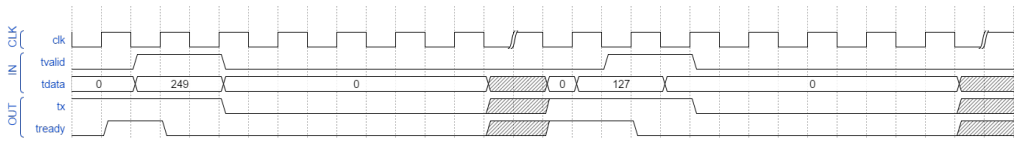

Figure 5.6: Wave traces generated by WaveDrom from code 5.14

This relatively small file forms the base of a testbench that will run for over 8000 clock periods. The wave traces are also shown in appendix B.1.1 in more detail.

5.3 Overview

The full code can be found online [11]. An overview of the code functionality is given in image 5.7. This overview corresponds to the purple collored section of figure 4.1

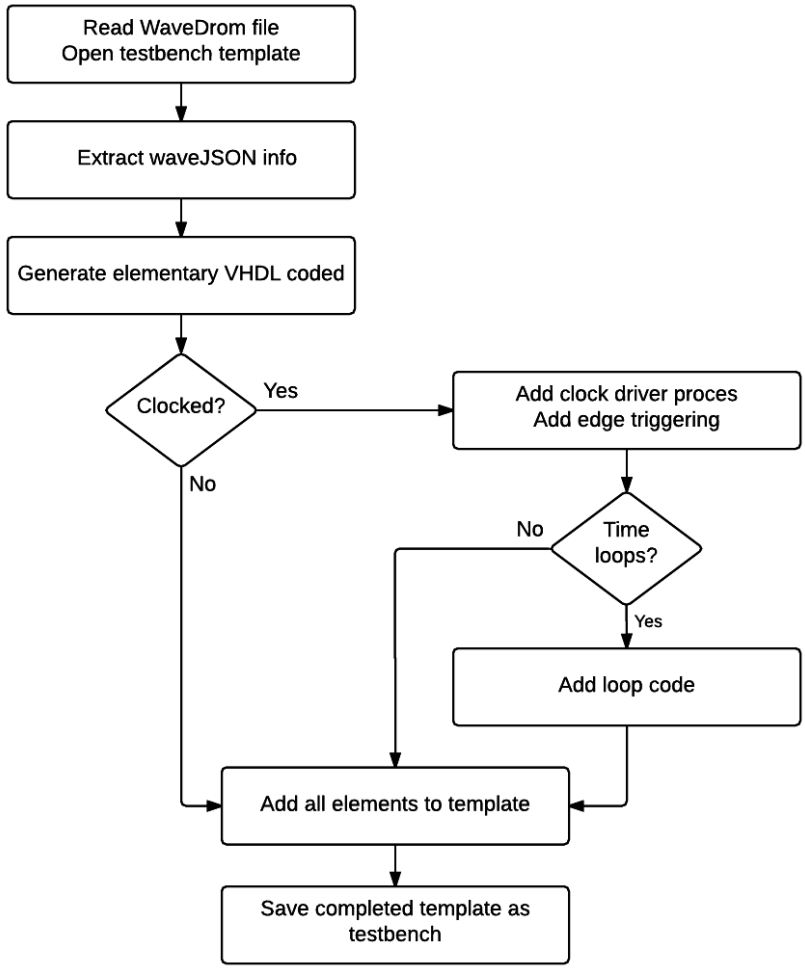

Figure 5.7: Overview of the testbench creation code

The code is run for every WaveDrom file in the resource folder (see appendix A: user guide). The method used in this program can be compared to building a prefabricated house. First the walls, cellar, floors and roof are fabricated in the factory. These are elements a basic house needs (combinatorial testbenches). Some houses require special additions like a swimming pool or solar panels, however (timed testbenches, looping testbenches). These will be fabricated only if necessary. Afterwards all elements will be put together in accordance with the blueprint (template).

From the WaveDrom file all info is extracted and all elementary testbench building blocks are generated. These are the elements every testbench needs. They include signal declarations, uut declaration and uut port map. A full list can be found in §5.1.1. The testbench header is also created. This holds the test name and description and will be added to the testbench later as a comment.

At this point all the building blocks to create a combinatorial testbench (a basic house) are available, but timed testbenches require some additions, like a clock driving process and edge triggering. To check for a timed test it suffices to check whether there is a signal labeled "CLK". If there is no "CLK" label, the test will be combinatorial and the testbench can be assembled. Otherwise the additional building blocks must be generated before the testbench can be assembled.

The building blocks available now are enough to build a timed testbench, but not testbenches that have time loops. If the "loop_times" field is defined in the clock signal, it means that there are time loops in this test and the corresponding building block should be generated too.

When all building block are generated they are added to the template using keywords. These keywords can be considered the blueprint of the house. They assign a place to every building block. When every building block is placed at the right keyword the testbench has been completed and the file can be saved.

5.4 Wave trace comparison

Until now the focus has been primarily on creating testbenches from WaveDrom input files. However, this is only part of the solution. In a design validation process the simulation result (wave traces) has to be compared to the description in the designs documentation.

Because the testbenches are directly derived from the same files that generate the documentation wave traces, it is safe to assume that full specification compatibility can be checked inside the testbench. In other words: if a test passes the simulation, it can be regarded as compliant to that aspect of its specification. In the same way it is certain that the current design does not comply if the test fails. Testbenches that meet these requirements are considered self checking. If that is the case, it is important to clearly determine where the simulation fails in order to understand what caused the defect and eventually correct it.

5.4.1 Ideal solution

As it is not desirable to stop a simulation when an error is encountered, errors should be logged during simulation and processed afterwards. As the simulation result and the documentation both consist of wave traces of the same signals, a simple and clear way to compare them would be to place each signal and their corresponding simulated signal side by side and place a marker to show every error that was logged during simulation. This principle is illustrated by the image below.

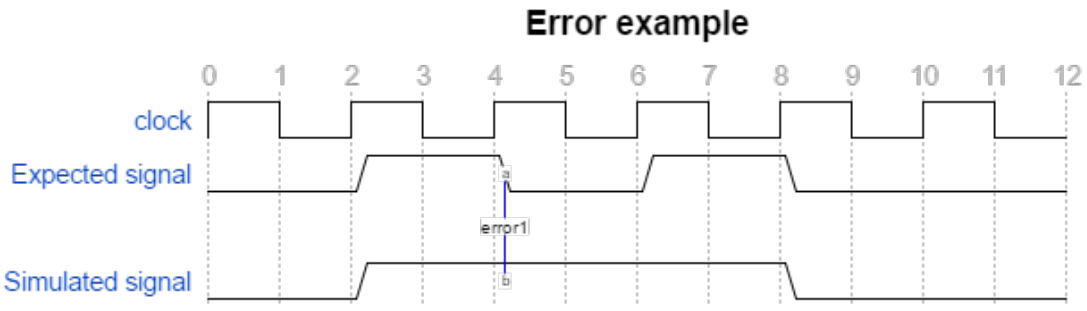

Figure 5.8: Ideal error marking

Every marker will be accompanied by a log message.

error1: unexpected output at clock tick 3, expected '1', got '0'.

At this point testbenches can be automatically generated by the software. In these testbenches every time step is numbered. This means that in the VHDL testbench there is an internal signal called 'n' that is incremented every internal clock cycle. Because the actual clock signal is driven on the rising and falling edge of the internal clock, there is a direct correspondence between 'n' and the x-th clock cycle in the wave trace file, where

$$x = n/period \tag{5.2}$$

and period is the relative clock period set by the "period" field in the WaveDrom file. In other words, n times the relative clock period gives a user the clock cycle on which an error occurred. The step number n is always displayed at the top in a simulation result.

To keep this correlation between wave trace file and simulation file we need to add one exception to the rule: n can only be incremented once during a time loop, because while a time loop can be hundreds of internal clock periods long, it is denoted as only one clock period in a wavedrom file.

5.4.2 Practical approach

Comparing WaveDrom generated wave traces and simulation generated wave traces remains a cumbersome process. When an error occurs a user still has to manually find the corresponding time step in both the documentation wave traces and the simulation wave traces and compare those.

The ideal system proposed in 5.4.1 could facilitate this process enormously. It would bundle the documentation wave traces and the simulation wave traces in one easy to comprehend wave trace file.

Because the documentation wave traces and the simulation wave traces are indirectly linked via the source file, it is possible to use the source file as a base for the result files and add the error messages logged during simulation to it. The VUnit Log package can log all error messages to a file. This log package is integrated into the VUnit Check package that offers conditional log messages, which is ideal for this

application. The code below is an example check for the signal sig_example. Because no specific checker is specified, the universal checker is automatically used.

```
if sig_example_values(n) /= 'X' then
  check(sig_example = sig_example_values(n),
  "This check failed. Expected example = " &
  std_logic'image(sig_example_values(n)) & ",
  got example = " & std_logic'image(sig_example) &
  " at n = " & integer'image(n) & ".");
end if;
```

Code 5.15: Signal checking in VHDL

If a signal is to be checked (the expected value is not 'X') the current value is compared to the expected value and an error report is generated when they are not equal. An example error report is shown in log message 2. The code below initialises the universal checker and is executed on testbench start.

```
checker_init(warning, "", ".warning_log/test_name_messages.csv",
    level, verbose_csv, failure, ',', false);
```

The initialisation process defines an output file for the log message. This way the log message from the check in code 5.15 is added to a .csv file in a hidden subfolder named ".warning_log/" and contains the entity name of the testbench it is derived from. An example message that could be logged to the file ".warning_log/example_messages.csv" by the example above is:

Warning: "This check failed. Expected example = '0',
got example = '1' at n = 4."

Log message 2: Example error message

This directly links the actual error to a specific point in time. From the current system it is possible to derive the system proposed as the ideal solution in §5.4.1.

Look and feel

The image below is a proposal for the layout of the comparison file based on a failing AND-gate test. More detail on this test can be found in §??.

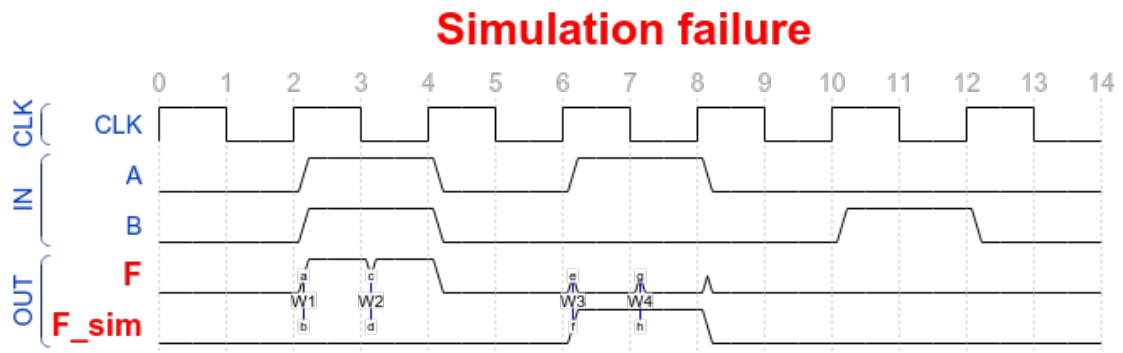

Figure 5.9: Layout proposal for result files

This image shows the input wave traces of signals 'A' and 'B', the expected wave traces of signal 'F' and the simulated wave traces of signal 'F_sim'. Every difference between expected and simulated signal wave traces is highlighted and labeled.

It is generated by a WaveDrom file very similar to the input file for this specific test software. This means that the input file can be easily manipulated by the software to create this image.

These changes include adding a title and an internal clock counter (equivalent to the 'n' signal in the simulation wave traces) in the header field, adding the simulation signal (F_sim) to the file and adding nodes and edges to the signals.

5.4.3 Preparations

The image in the previous section was generated by the following WaveDrom file, while the original code for the failing AND-gate test can be found in §??.

```
{"name": "andGate_timed", "test" : "andgate_failing",
"description": "a full AND-gate test designed to fail",  "signal"
   : [
  ["CLK",
    {"name": "CLK", "wave": "p......", "type":"std_logic", "
   period":"2", "clock_period": "20"}],
  ["IN",
    {"name": "A", "wave": "01010..", "type": "std_logic", "period
   ":"2"},
    {"name": "B", "wave": "0.1.0.....1.0.", "type": "std_logic"}]
   ,
  ["OUT",
    {"name": "F",      "wave": "0.....1.0.....", "type": "
   std_logic", "node": "..ac..eg"},
    {"name": "F_sim", "wave": "0.110.000.....", "type": "
   std_logic", "node": "..bd..fh"}]
],
"head": {"text": ["tspan", {"class": "error h3"}, "Simulation
   failure "], "tick": 0},
"edge": ["a-b W1", "c-d W2", "e-f W3", "g-h W4"]
}
```

Code 5.16: Source file for the proposed layout in fig 5.9

The failing AND-gate test will output the following messages:

WARNING: This check failed. Expected F = '0', got F = '1' at n = 2.

WARNING: This check failed. Expected F = '0', got F = '1' at n = 3.

WARNING: This check failed. Expected F = '1', got F = '0' at n = 6.

WARNING: This check failed. Expected F = '1', got F = '0' at n = 7.

Log message 3: Failing andgate log messages

The original code serves as the base for the final comparison file. Additional information can be added based on the log messages above.

In a coding environment this is not an information format that is easy to work with, so another log file is created, which will hold the same information in a more processable format. To write to this new file, a new checker is added to every testbench. This is done by adding a new variable of type checker_t to the template file (which will cause it to be added to every generated testbench) and initializing it similarly to the universal checker (see code 5.4.2).

```
shared variable warning_logger : checker_t;
```

Code 5.17: Custom checker declaration

```
checker_init(warning_logger, warning, "", ".warning_log/
    test_name_logs.csv", level, verbose_csv, failure, ',', false);
```

Code 5.18: Custom checker initialisation

Now a new conditional log can be added that will write to the new file. The full check code for an example signal is now:

```
if sig_example_values(n) /= 'X' then
  check(sig_example = sig_example_values(n),
    integer'image(error_number) & "original message"); --
    original message is logged here
  check(warning_logger, sig_example = sig_example_values(n),
    integer'image(error_number) & "new log message", line_num =>
    error_number); -- new log message is logged here
  if sig_example /= sig_example_values(n) then
    error_number := error_number +1;
  end if;
end if;
```

Code 5.19: Improved signal checking

Where the universal checker is used to conditionally log the original message and the new warning_logger checker is used to log the new message to another file. To link the original messages to the new logs, an error number is added to both logs, which is incremented every time an error occurs.

The new warning logs are made following this pattern:

["warning no", "signal involved","expected value", "actual value", "n"]

The full output after running the failing AND-gate example will now be:

WARNING: 1. This check failed. Expected F = '0', got F = '1' at n = 2.

WARNING: 2. This check failed. Expected F = '0', got F = '1' at n = 3.

WARNING: 3. This check failed. Expected F = '1', got F = '0' at n = 6.

WARNING: 4. This check failed. Expected F = '1', got F = '0' at n = 7.

Log message 4: Log messages in the "andgate_failing_message.csv" file

["1", "F", "0", "1", "2"]

["2", "F", "0", "1", "3"]

["3", "F", "1", "0", "6"]

["4", "F", "1", "0", "7"]

Log message 5: Log messages in the "andgate_failing_log.csv" file

Now all information can be easily imported and processed. The software can read both the WaveDrom JSON file and the logged information, modify the JSON content and write it to an output file.

5.4.4 WaveDrom result file

As stated in §5.4.2 the result file can be generated from the original input file and the warning logs (the "test_name.json" and the "test_name_result.csv" in the relative directory "sresult"). The flow diagram below shows the function of the software that generates the compare files. This image corresponds to the green part of figure 4.1.

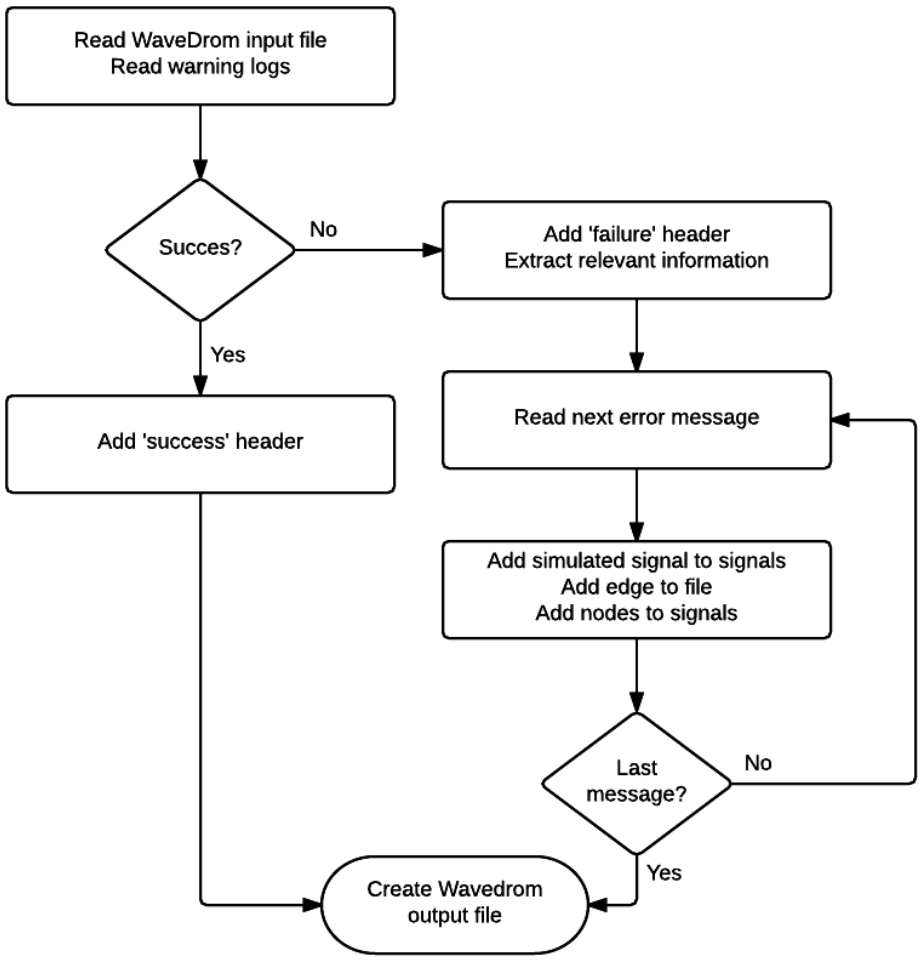

Figure 5.10: Overview of the wave trace analysis code

After reading the input files, the software checks whether or not there are messages logged. If there are no logged messages, the simulation succeeded. In this case there is nothing else to be done than to add a header and create the output file. In case there are messages logged, however, the simulation failed and the process becomes more complicated.A corresponding header is added containing a title and the time step counter. Then all messages are read one by one and further processed.

Consider the WaveDrom input file and the output messages from the failing AND-gate example:

```
{"name": "andGate_timed", "test" : "andgate_failing",
"description": "a full AND-gate test designed to fail",
"signal": [
  ["CLK",
    {"name": "CLK", "wave": "p......", "type":"std_logic", "
   period":"2"}],
  ["IN",
    {"name": "A", "wave": "0.1.0.1.0.....", "type": "std_logic"},
    {"name": "B", "wave": "0.1.0.....1.0.", "type": "std_logic"}]
   ,
  ["OUT",
    {"name": "F", "wave": "0.....1.0.....", "type": "std_logic"}]
]}
```

Code 5.20: Source file for a failing AND-gate example

[”1”, ”F”, ”0”, ”1”, ”2”]

[”2”, ”F”, ”0”, ”1”, ”3”]

[”3”, ”F”, ”1”, ”0”, ”6”]

[”4”, ”F”, ”1”, ”0”, ”7”]

Log message 6: Log messages in the ”andgate_failing_result.csv” file

Each message alters the output file in several ways. Consider the first message for example. When the software reads a new message, it first converts the message to a readable list and then determines the name of the signal involved, which is 'F'. It then finds that signal in the current WaveDrom file and copies it. This copy is renamed to 'F_sim' and added to the output data. The wave traces for this signal are exactly equal to the wave traces of 'F', which means that it holds the same errors. The errors is the 'F' signal are fixed by replacing the wave value at time step 'n', given in the last field by the 'expected value' in the second to last field.

Then, for both the 'F' and 'F_sim' signal, the "node" field is added. This field adds a name to a specific time step in a signal. This is necessary to enable the next step. The node field for the 'F' and 'F_sim' signals would be:

```
1   "node": "..a"
2   "node": "..b"
```

respectively. This way, for both signals, the third time step (time steps are numbered starting from 0) has a name. A named time step is also called a node.

Finally, a line, called an edge, is added to connect both nodes. This is done by appending an edge field to the WaveDrom file.

```
1 "edge":["a-b W1"]
```

This way an edge labeled 'W1' connecting node a to b will be created. This label is derived from the error number in the first field of the log message.

Lastly, all signals involved ('F' and 'F_sim') will be marked red.

The JSON data imported from the source file will now look like this:

```
{"name": "andGate_timed", "test" : "andgate_full", "description":
    "Test all possible inputs for an AND-gate", "signal": [
  ["CLK",
    {"name": "CLK", "wave": "p......", "type":"std_logic", "
   period":"2", "clock_period": "20"}],
  ["IN",
    {"name": "A", "wave": "01010..", "type": "std_logic", "period
   ":"2"},
    {"name": "B", "wave": "0.1.0.....1.0.", "type": "std_logic"}]
   ,
  ["OUT",
    {"name": "F",      "wave": ""wave": "0.10..1.0.....", "type":
   "std_logic", "node": "..a"},
    {"name": "F_sim", "wave": ""wave": "0.....1.0.....", "type":
   "std_logic", "node": "..b"}]
], "head": {"text": ["tspan", {"class": "error h3"}, "Simulation
    failure "], "tick": 0},
"edge": ["a-b W1"]
}
```

Code 5.21: Temporary content of the result file of a failing AND-gate example

Which yields:

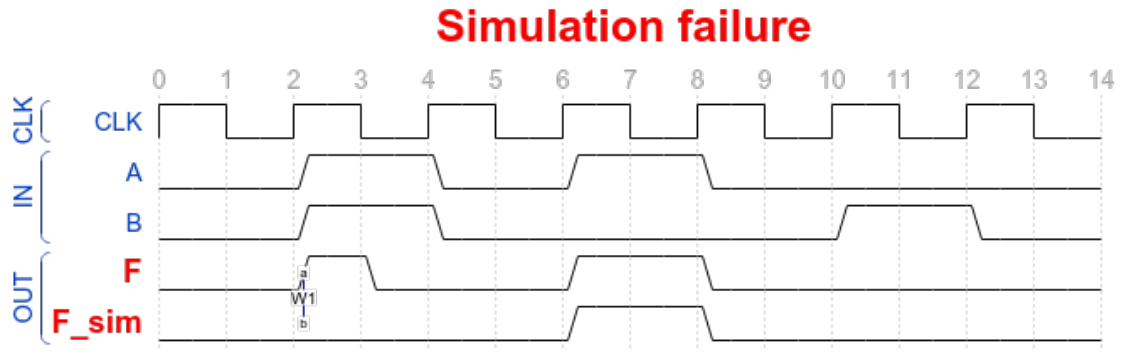

Figure 5.11: Wave traces generated by WaveDrom from code 5.21

After this, the second message is read and processed the same way. The only difference is that 'F_sim' already exists and does not have to be added again.

When all four messages are processed the JSON data is written to a file in the ouput folder. The file is shown in code 5.22.

```
{"name": "andGate_timed", "test" : "andgate_failing",
"description": "a full AND-gate test designed to fail",
"signal": [
  ["CLK",
    {"name": "CLK", "wave": "p......", "type":"std_logic", "
   period":"2", "clock_period": "20"}],
  ["IN",
    {"name": "A", "wave": "01010..", "type": "std_logic", "period
   ":"2"},
    {"name": "B", "wave": "0.1.0.....1.0.", "type": "std_logic"}]
  ,["OUT",
    {"name": "F",      "wave": "0.....1.0.....", "type": "
   std_logic", "node": "..ac..eg"},
    {"name": "F_sim", "wave": "0.110.000.....", "type": "
   std_logic", "node": "..bd..fh"}]
  ],
"head": {"text": ["tspan", {"class": "error h3"}, "Simulation
    failure "], "tick": 0},
"edge": ["a-b W1", "c-d W2", "e-f W3", "g-h W4"]
}
```

Code 5.22: Final content of the result file of a failing AND-gate example

This yields figure 5.12.

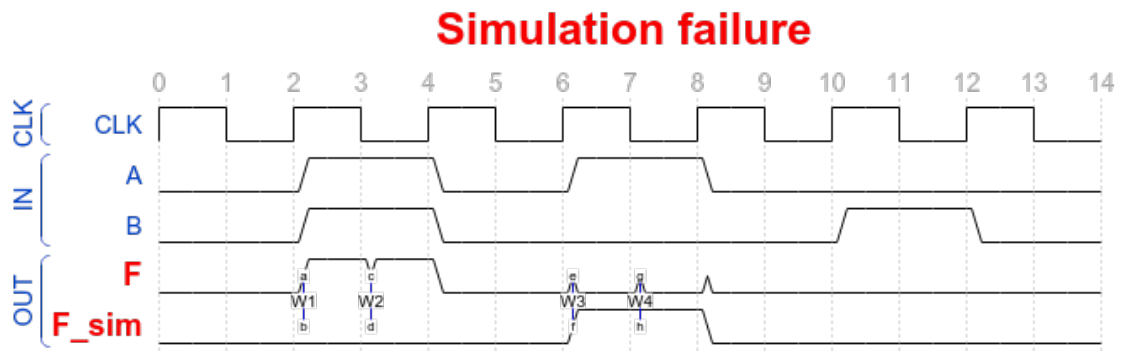

Figure 5.12: Wave traces generated by WaveDrom from code 5.22

Which is exactly the same as the goal image in figure 5.9.

5.5 Limitations & expansions

This chapter lists the main limitations of the current design and proposes some improvements that could benefit performance and usability.

5.5.1 Difficult to install

Although the system is quite simple to use once installed, it is not easy to install. Because of the dependencies of the system and their respective dependencies it requires some skill to set up the system. This is all the more so because it was developed on a Linux system and is not directly compatible with Windows computers. Compatibility with OS X computers has not been tested.

5.5.2 Generic input values

Consider the shift register example at **??**. This example tests the design with the input sequence of I = "0001101111" with the shift input enabled. Suppose that the designer wants to build another test with the same input, but with the shift input disabled. This would mean that he will have to run the software with a new WaveDrom file simply to change this one value to create a whole new testbench. In this case making another testbench seems doable, considering that the shift enable input has only two values, but as the level of complexity goes up the amount of testbenches rises.

Suppose another designer is building a memory unit and would like to build a test that writes and then reads every memory element. At this moment there is no way to build tests that will do this without the designer having to make an input file for every memory address.

An answer to this problem could lie in the possibility to dynamically generate JSON signal declarations as shown in the official WaveDrom tutorial under "step 9. Some code".

5.5.3 Simulator compatibility

During development of the system the Vsim simulator from Modelsim was used. On the same system the GHDL open source simulator was also tested, but this resulted in unresolved simulation errors. This means that not all simulator backends supported by VUnit are compatible with this software.

5.5.4 Redundant data

In the beginning of this report the assumption was made that all information necessary to create testbenches was included in the WaveDrom file. Because of this some major extensions have been added to that file. Almost all of this information, such as signal direction, logic type and vector width, is however already available in the original design file and is actually redundant. Accessing this data in the design file is more difficult and requires the use of a VHDL parser, which is why this approach was not used in this project.

It would, however, be a considerable improvement to change the way the relevant data is acquired. The greatest benefit being that a lot of data could be removed from the WaveDrom files to make them more readable and easier to create.

5.5.5 Result readability

Looking at the examples in **??** we see an increasing complexity of the UUT design. This means that more and more information has to be added to the WaveDrom files. Correspondingly, the wave traces that are generated become increasingly complex to the point where they can not be properly displayed on one page. This limits the readability of the resulting wave traces.

Larger designs mean more signals to drive and a more complex way of functioning, which might also mean larger WaveDrom input files. This is especially true if the output signals change often and independently from each other. Larger WaveDrom files mean longer wave traces.

If a test's wave traces can not be displayed on a single page, they might not be fit to be used as documentation as they might incomprehensible. This problem is illustrated by example B.1.2.

5.5.6 CoCoTB vs VUnit

Every testbench in the current approach is based on the VUnit framework. VUnit is in essence an extension of the existing VHDL/Verilog languages with a Python framework built around it. This Python framework makes it possible to run testbenches using an external simulator backend.

CoCoTB [4] is also an environment for verifying Verilog and VHDL using python. It is similar to VUnit, but the way testbenches are created differs greatly. CoCoTB offers a way to write testbenches in python, while VUnit testbenches are created in VHDL or Verilog.

In this sense CoCoTB holds an advantage over VUnit. As VHDL and Verilog are more static languages only used by few, Python is widely used and constantly in development and has an large and active community, which means there is an enormous amount of information available for it. Writing testbenches in Python rather than in hardware description languages can also help to make them easier to understand. In other aspects CoCoTB offers the same features as VUnit such as a logging system, easy signal checking methods, multiple simulator backend support, unit testing etc. .

In general, it seems that switching to CoCoTB as the environment for testbenches could be an enhancement for the project in terms of readability, ease of use and ease of development.

Chapter 6

Conclusion

In light of improving hardware design verification this report has tried to answer the following questions:

- Can we design a system that will streamline the process of design validation?

  Can we design a system that can create self-checking testbenches and documentation wave traces from the same source?

  Can we optimise wave trace analysis based on this system?

- Does this system provide any benefits over existing verification frameworks?

The sections below will recapitulate the solution provided in this report and answer these questions.

6.1 Self checking testbenches and documentation wave traces

The basic assumption at the start of the project was that it should be possible to create both documentation wave traces and testbenches from the same file. Finding the correct format for this file was imperative. The solution proposed by the promoters of this project seemed ideal. WaveDrom can draw a timing diagram (or wave traces) from a simple description. This description is provided in a waveJSON format. Using WaveDrom, half of the desired functionality was obtained. The second half, creating self-checking testbenches, was made possible by adding some extra required information to the original WaveDrom file. The adapted WaveDrom

file is now considered a complete source file and holds all data needed to create a testbench for a specific design. It is a compact and simple way to hold all of the required information.

As there was no tool that could create test benches from the source file yet, it had to be developed. Based on some increasingly complex design examples, Python based software was developed that can translate the info in the source file into a fully fledged self-checking testbench. This testbench satisfies the conditions set by the VUnit framework.

The resulting software can turn every source file that satisfies the right conditions (see user guide) into a VHDL self-checking testbench for a specific design, while WaveDrom can generate documentation wave traces from this same file. This means that we have indeed created a system that can create self-checking testbenches and documentation wave traces from the same source.

6.2 Optimizing wave trace analysis

Now that a system exists that can create testbenches, the next step was to check whether or not the tests performed in these testbenches were successful. In other words, whether or not the design behaved as expected.

The VUnit framework can help simulate the previously mentioned testbenches and can log simulation information to a file. Using this functionality all simulation error messages were logged. These messages are only generated if there is irregular functionality in the design file. It contains all the information regarding the errors during the simulation. These messages were used to create a new file, which is basically an adaptation of the source file. When this source file is converted to wave traces by WaveDrom it clearly shows where the simulation output differs from the wave traces generated by the original source file. An accompanying text file provides textual information on the differences. Using these output wave traces and the output text file a user can immediately see what went wrong during simulation.

As wave trace analysis used to be a matter of visually comparing documentation wave traces to the simulation wave traces created by a simulator, we can say that this process is indeed an optimization of the wave trace analysis.

6.3 Streamlining the process of validation

We have facilitated the process of wave trace analysis and testbench creation, while at the same time minimizing user input by combining documentation wave trace creation with testbench creation. In other words we have designed a system that transformed the original validation process, where documentation wave traces had to be created separately and testbench creation and wave trace analysis had to be done manually, into a system where these things are automated and user input has been minimized The process flow went from figure 6.1 to figure 6.2.

Considering the limitations and expansions mentioned in §5.5 this system is not yet reached its full potential, but we can still say we have at least laid the foundation for streamlining the validation process.

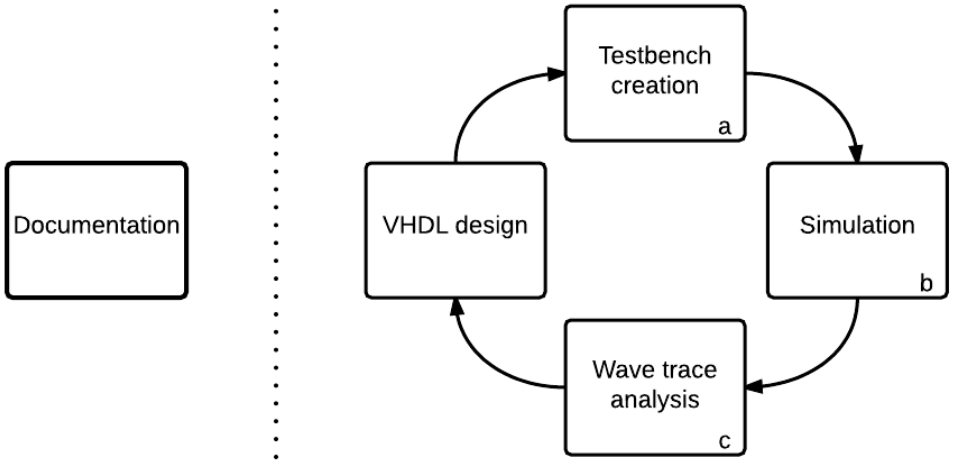

Figure 6.1: Traditional verification cycle

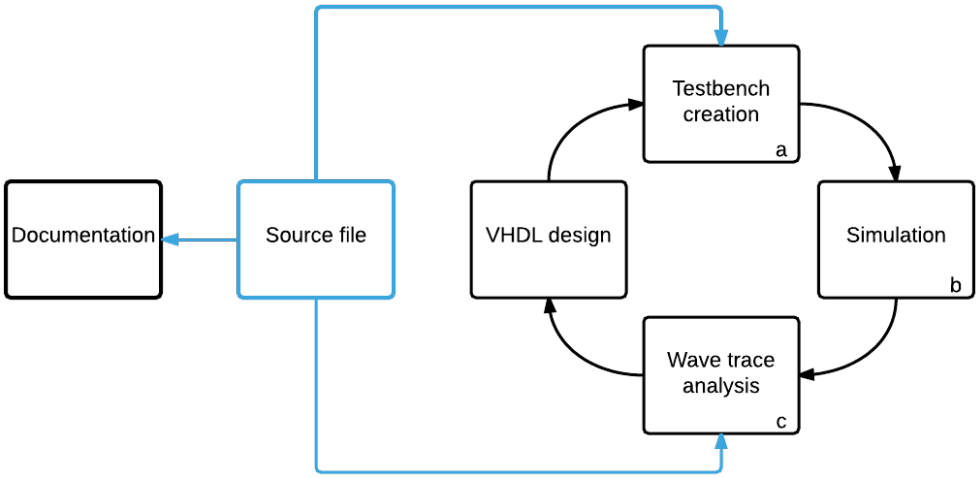

Figure 6.2: Improved verification cycle

6.4 Benefits over existing verification frameworks

Current frameworks such as VUnit and CoCoTB offer an extension of the hardware design languages in the form of additional packages and provide a (Python) framework to integrate the running of tests in other software. As the system that has been developed builds on this to provide extra functionality and ease of use, we can say that it could indeed prove beneficial to users with certain needs. Because of the limitations discussed in §5.5 however the current version of the system also limits the original functionality of these frameworks.

High end users might choose better coverage over automated wave trace comparison or another backend simulator over automated testbench creation and will not use the tool in its current form. On the other hand novice users might appreciate the ease of use and WYSIWYG (What You See Is What You Get) approach of the source file and the clear error messages over complicated, but more powerful validation methods.

6.5 Fields of application

These different preferences limit the scope of use for this tool as it is now. Installing it requires an experienced user, but those experienced user are often also more high end users, who might not benefit from using this system. Therefore the most important field of application might be education. The system is ideal for students who are new to HDL design and are only starting to describe their own hardware components. They could start to validate their designs using self made WaveDrom files and generate documentation for a report at the same time.

Once the amount of supported frameworks and simulator backends grows, the system could also be used to perform general validation tests in more high end environments, while full coverage would be attained by using conventional methods.

Bibliography

[1] "Software verification and validation." [Online]. Available: \en.wikipedia.org/wiki/Software_verification_and_validation

[2] "Wavedrom homepage." [Online]. Available: wavedrom.com

[3] "Vunit home page." [Online]. Available: vunit.github.io

[4] "Cocotb documentation." [Online]. Available: cocotb.readthedocs.io/

[5] "Svunit home page." [Online]. Available: http://www.agilesoc.com/open-source-projects/svunit/

[6] "Osvvm home page." [Online]. Available: http://osvvm.org/

[7] "Uvvm home page." [Online]. Available: http://bitvis.no/products/uvvm-utility-library/

[8] "Vunit home page." [Online]. Available: vunit.github.io/documentation

[9] "wavejson data interchange format." [Online]. Available: \https://github.com/drom/wavedrom/wiki/WaveJSON

[10] "Online examples for this project." [Online]. Available: \https://github.com/WinandS/Thesis/tree/master/examples

[11] "Project source repository." [Online]. Available: github.ugent.be/wseldesl/Thesis

[12] "Project user guide." [Online]. Available: \github.ugent.be/wseldesl/Thesis/blob/master/README.md

[13] "Python home page." [Online]. Available: www.python.org

[14] "Teahlab website." [Online]. Available: www.teahlab.com

[15] "Esd website." [Online]. Available: http://esd.cs.ucr.edu/labs/tutorial/

[16] "Spi master vhdl project." [Online]. Available: www.lothar-miller.de/s9y/categories/45-SPI-Master

Appendix A

User guide

This appendix contains the information needed to use the software. The user guide can also be found at [12]

A.1 Dependencies

The final product of this project is a software tool written in Python that can run on any computer that complies to these conditions:

- It runs a linux distribution (developed on Ubuntu 14.04 LTS)
- It has at least Python [13] 2.7 installed
- It has VUnit [8] installed
- It has WaveDrom [2] installed

A.2 project input files

A.2.1 Making your own WaveDrom input files

These guidelines will help users to create WaveDrom files compatible with the system. Normal operation is not guaranteed when these guidelines are not respected.

Standard input file

This project uses WaveDrom files as the base for its input file. Some fields have been added to the standard JSON file and should always be included as shown in the example below. Each WaveDrom file used with this software should contain these elements:

name(*): this should be the exact name of the unit under test

test: this is the name for this test

description: holds the descrition for this test

type: should be specified for every signal; The type specifies the VHDL logic type for this signal

vector_size: for vectors (like tdata in the example below) the vector size should also be specified. This is the amount of bits that the vector represents

data: The data that the vector holds should be specified in this list

clock_period: (optional) The designer can specify a clock signal, otherwise the clock period will default to 20 ns for each internal clock period

Next to these extra fields, each signal should be placed under the apropriate label. All input signals should be under one "IN" label, all output signals under one "OUT" label and the clock signal under the "CLK" label. The clock signal is regarded as a special case input signal and is treated seperately.

Unicode keys

Although WaveDrom supports non unicode keys. e.g.:

```
{period : 2}
```

The python JSON package used does not. Because of this all keys must be unicode strings. e.g.:

```
{"period" : 2}
```

Supported characters

Not all wavedrom characters are currently supported. The supported characters are:

In a clock signal:

```
{'n', 'p', '.', '|'}
```

In an input signal:

```
{'1', '0', '.', '='}
```

In an output signal:

```
{'1', '0', '.', '=', 'x'}
```

A.3 Extra features

A.3.1 time loops ('|')

It is possible to simulate a period where the input never changes. By using the '|' character in the clock signal and the 'loop_times' field a loop is defined. They specify the moment this period should start and the amount of clock cycles it should last. In the example above the first loop period starts at the eighth clock period and lasts for 10*434 clock periods. This is the time required for the design to send 10 bits over tx (one start and one stop bit, and the data from tdata sent sequentially). During this loop the checks at this moment are repeated every clock cycle. For obvious reasons this feature can only be used when a clock signal is present.

A.3.2 check skipping ('x')

In some cases the output at a specific point in time is undefined or is of no interest to the designer. In this case the designer can use the 'x' character, which means that the signal should not be checked at this point. This character is only valid for output signals.

A.4 Example

Here is an example input file testing the design of an UART transmitter design "uart_tx" available in appendix C.3. Two consecutive bytes are sent over the serial output port. Input signals as well as expected output signals are defined.

```
{"name": "uart_tx", "test" : "uart_send_2_bytes", "description": "A test for sending two consecutive bytes with an parallel to serial uart", "signal": [
  ["CLK",
    {"name": "clk", "wave": "n......|n......|", "type":"std_logic", "period": "2", "clock_period": "10", "loop_times" : ["10*434", "10*434"]}],
  ["IN",
    {"name": "tvalid", "wave": "0.1..0.............1..0.........", "type": "std_logic"},
    {"name" : "tdata", "wave": "=.=..=........x.==..=........x.", "data": ["0", "249", "0", "0", "127", "0"], "type" : "std_logic_vector", "vector_size" : "8"}],
  ["OUT",
    {"name": "tx", "wave": "1....0........x.1....0........x.", "type": "std_logic"},
    {"name": "tready", "wave": "01.0.........x.1..0.........x.", "type": "std_logic"}]
]}
```

The generated wave trace file can be found in appendix B.1.1.

A.5 Starting the tool

1. go to https://github.ugent.be/wseldesl/Thesis
2. Make sure you have met all requirements for running the program
3. Download the software as a zip file
4. Extract the zip file
5. Go to the Code folder
6. Run main.py
7. Click 'open input folder' and add input files
8. Click 'open design folder' and add design file to test
9. Click start button
10. Click 'open output folder'
11. Open result waveJSON files in WaveDrom and corresponding text file for error messages

Appendix B

More examples

The verification examples in **??** are continued here. Each test will be preceded by a short description, followed by the Wavedrom input code and its corresponding generated wave traces image. If the test fails, the result wave traces and corresponding wave traces will also be shown, but for successful tests the generated documentation wave traces will not be shown as they are the same as the resulting output wave traces.

B.1 Successful tests

B.1.1 UART

An UART transmit line design is tested by trying to send two bytes of data in sequence. This test is designed to check whether it is possible to send multiple bytes in sequence and ignores the actual sending of the byte.

```
{"name": "uart_tx", "test" : "uart_send_2_bytes", "description": "A test for sending two
    consecutive bytes with an parallel to serial uart", "signal": [
["CLK",
  {"name": "clk", "wave": "n......|n......|", "type":"std_logic", "period": "2", "clock_period":
    "10", "loop_times" : ["10*434", "10*434"]}],
["IN",
  {"name": "tvalid", "wave": "0.1..0...........1..0.........", "type": "std_logic"},
  {"name" : "tdata", "wave": "=.=..=........x.==..=........x.", "data": ["0", "249", "0", "0",
    "127", "0"], "type" : "std_logic_vector", "vector_size" : "8"}],
["OUT",
  {"name": "tx", "wave": "1....0........x.1....0........x.", "type": "std_logic"},
  {"name": "tready", "wave": "01.0.........x.1..0.........x.", "type": "std_logic"}]
]}
```

Code B.1: Functionality test for the UART design in appendix C.3

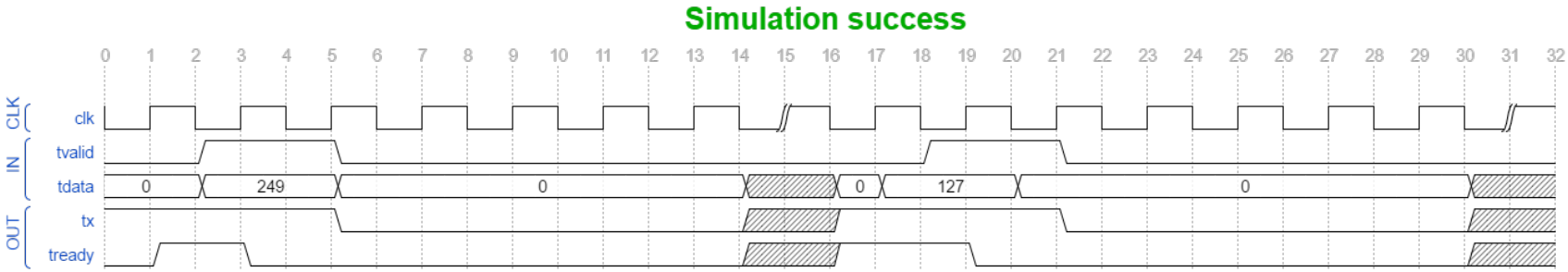

Figure B.1: Simulation result for the test described in code B.1

B.1.2 SPI

Similarly to the previous UART example this example shows an SPI master sending a byte to a slave. The main difference here is that the sending of the bits is not skipped using the 'x' character. Instead every change in output is checked. The wave traces generated by this source are so large they become difficult to depict. This shows the limits of this representation.

```
{"name": "SPI_Master", "test" : "SPI_send_one_byte",
"description": "A test for an SPI master sending one byte over an SPI connection",
"signal": [
  ["CLK",
    {"name": "clk", "wave": "p..|p.||p.||||||||||||p|||p...", "period" : "2", "type":"std_logic"
   , "loop_times" : ["150", "25", "24", "24", "25", "25", "25", "25", "25", "25", "25", "25", "2
   5", "25", "25", "24", "25", "25"]}],
  ["IN",
    {"name": "TX_Data", "wave": "=.....................................................",
    "data" : ["126"],"vector_size" : "8" , "type": "std_logic_vector"},
    {"name": "MISO", "wave": "0.....................................................", "
   type": "std_logic"},
    {"name": "TX_Start", "wave": "0.........1.....................................0.....",
    "type": "std_logic"}],
  ["OUT",
    {"name": "RX_Data", "wave": "=.....................................................",
    "data" : ["0"],"vector_size" : "8", "type": "std_logic_vector"},
    {"name": "MOSI", "wave": "0...................1.......................0............", "
   type": "std_logic"},
    {"name": "SCLK", "wave": "x.0..........1...0...1.0.1.0.1.0.1.0.1.0.1.0...1.0.........", "
   type": "std_logic"},
    {"name": "SS", "wave": "x.1.........0.....................................1.....", "type
   ": "std_logic"},
    {"name": "TX_Done", "wave": "x.0.....................................................1.0...",
   "type": "std_logic"}]]
}
```

Code B.2: Functionality test for the SPI design in appendix C.4

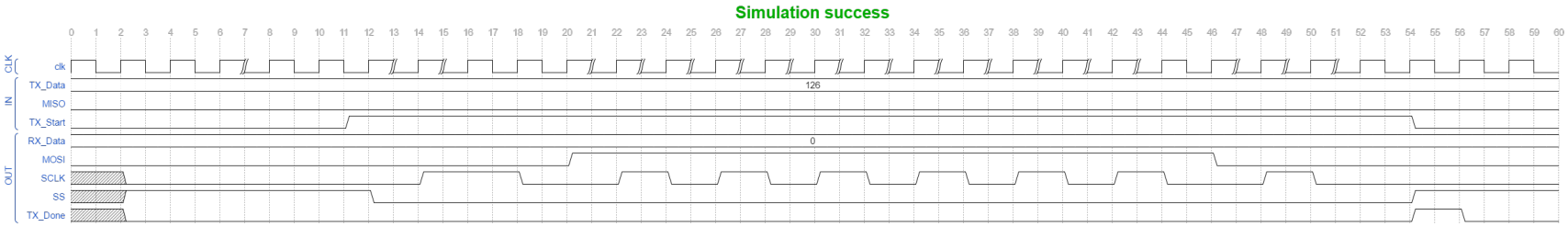

Figure B.2: Simulation result for the test described in code B.2

B.2 Failing tests

B.2.1 UART

This test is designed for the same purpose as the previous UART test, except that several output signals are expected to be different.

```
{"name": "uart_tx", "test" : "uart_send_2_bytes_failing", "description": "A test for sending two
    consecutive bytes with an parallel to serial uart designed to fail", "signal": [
  ["CLK",
    {"name": "clk", "wave": "n......|n......|", "type":"std_logic", "period": "2", "clock_period
   ": "10", "loop_times" : ["10*434", "10*434"]}],
  ["IN",
    {"name": "tvalid", "wave": "0.1..0...........1..0.........", "type": "std_logic"},
    {"name" : "tdata", "wave": "=.=..=........x.==..=........x.", "data": ["0", "249", "0", "0"
   , "127", "0"], "type" : "std_logic_vector", "vector_size" : "8"}],
  ["OUT",
    {"name": "tx", "wave": "1.....0.......x.1.....0.......x.", "type": "std_logic"},
  {"name": "tready", "wave": "01...0........x.1..0.........x.", "type": "std_logic"}]
]}
```

Code B.3: Failing functionality test for the UART design in appendix C.3

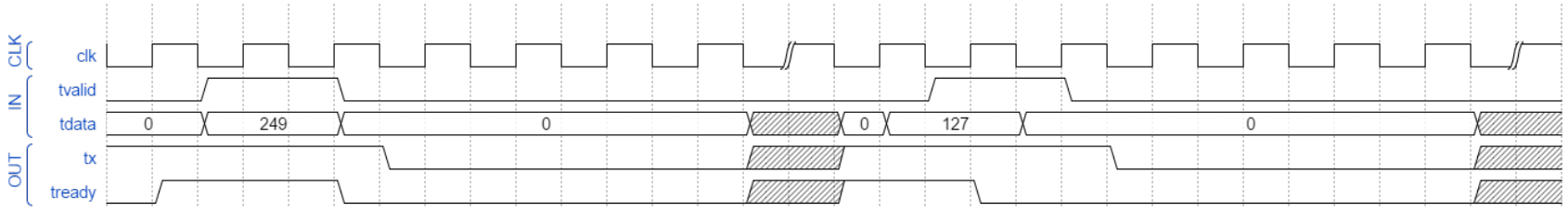

Figure B.3: WaveDrom generated documentation wave traces based on code B.3

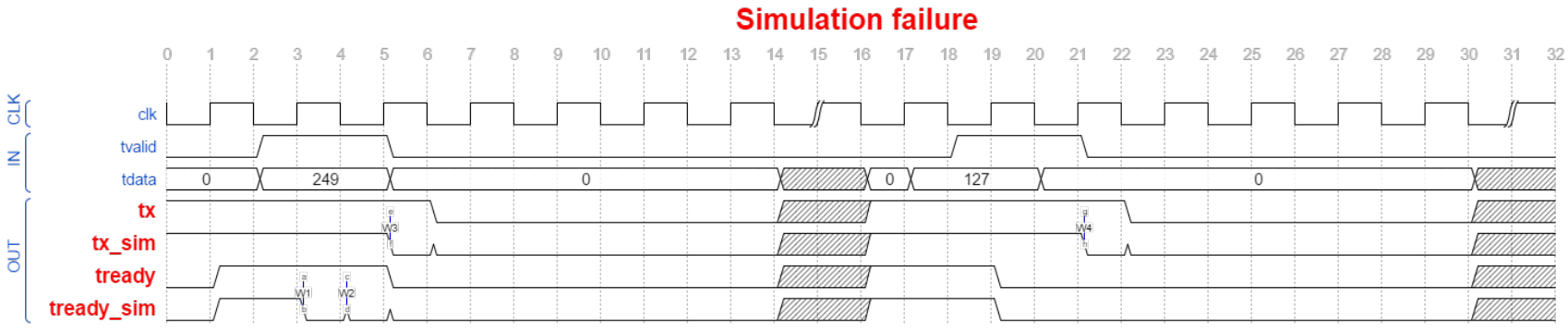

Figure B.4: Simulation result for the test described in code B.3

W1: Expected sig_tready = '1', got sig_tready = '0' at n = 3.

W2: Expected sig_tready = '1', got sig_tready = '0' at n = 4.

W3: Expected sig_tx = '1', got sig_tx = '0' at n = 5.

W4: Expected sig_tx = '1', got sig_tx = '0' at n = 21.

Log message 7: Log messages for the test in code B.3

B.2.2 SPI

This test is designed for the same purpose as the previous SPI test, except that several output signals are expected to be different.

```
{"name": "SPI_Master", "test" : "SPI_send_one_byte",
"description": "A test for an SPI master sending one byte over an SPI connection",
"signal": [
  ["CLK",
    {"name": "clk", "wave": "p..|p.||p.||||||||||||p|||p...", "period" : "2", "type":"std_logic"
   , "loop_times" : ["150", "25", "24", "24", "25", "25", "25", "25", "25", "25", "25", "25", "2
   5", "25", "25", "24", "25", "25"]}],
  ["IN",
    {"name": "TX_Data", "wave": "=.....................................................",
   "data" : ["126"],"vector_size" : "8" , "type": "std_logic_vector"},
    {"name": "MISO", "wave": "0.....................................................", "
   type": "std_logic"},
    {"name": "TX_Start", "wave": "0.........1.....................................0.....",
    "type": "std_logic"}],
  ["OUT",
    {"name": "RX_Data", "wave": "=.....................................................",
   "data" : ["0"],"vector_size" : "8", "type": "std_logic_vector"},
    {"name": "MOSI", "wave": "0.........10........1.......................0............", "
   type": "std_logic"},
    {"name": "SCLK", "wave": "x.0...10......1...0...1.0.1.0.1.0.1.0.1.0.1.0...1.0.........", "
   type": "std_logic"},
    {"name": "SS", "wave": "x.1.........0.....................................1.....", "type
   ": "std_logic"},
    {"name": "TX_Done", "wave": "x.010.....................................................1.0...",
   "type": "std_logic"}]
]}
```

Code B.4: Failing functionality test for the SPI design in appendix C.4

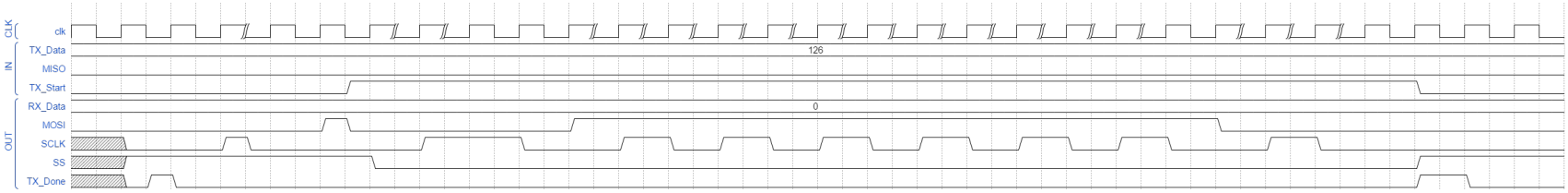

Figure B.5: WaveDrom generated documentation wave traces based on code B.4

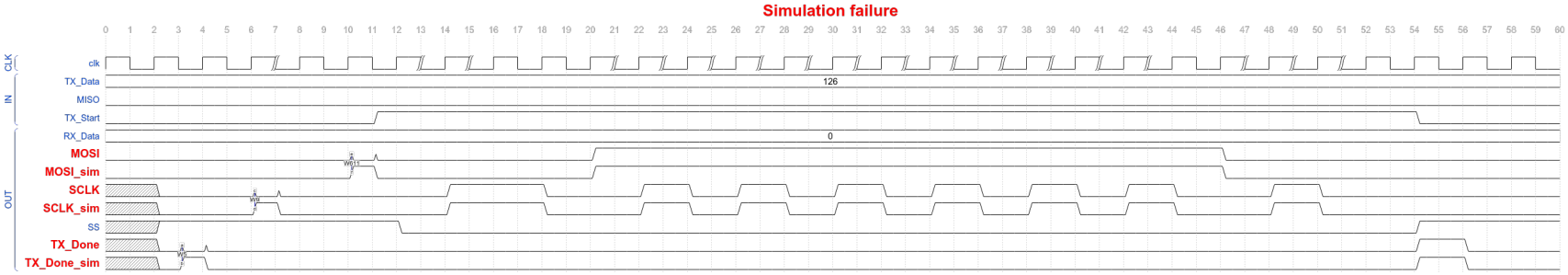

Figure B.6: Simulation result for the test described in code B.4

W5: Expected sig_TX_Done = '1', got sig_TX_Done = '0' at n = 3.

W9: Expected sig_SCLK = '1', got sig_SCLK = '0' at n = 6.

W611: Expected sig_MOSI = '1', got sig_MOSI = '0' at n = 10.

Log message 8: Log messages for the test in code B.4

Appendix C

Designs

This appendix holds the VHDL description of all hardware designs relevant to the report.

C.1 AND-gate

C.1.1 Combinatorial

```
--import std_logic from the IEEE library
library ieee;
use ieee.std_logic_1164.all;

--ENTITY DECLARATION: name, inputs, outputs
entity andGate is
   port( A, B : in std_logic;
         F : out std_logic);
end andGate;

--FUNCTIONAL DESCRIPTION: how the AND Gate works
architecture func of andGate is
begin
  F <= A and B;
end func;
```

Code C.1: Combinatorial AND-gate VHDL description downloaded from [14]

C.1.2 Timed or sequential

```
--import std_logic from the IEEE library
library ieee;
use ieee.std_logic_1164.all;

--ENTITY DECLARATION: name, inputs, outputs
entity andGate_timed is
  port(A, B, CLK : in  std_logic;
       F         : out std_logic := '0');
end andGate_timed;

--FUNCTIONAL DESCRIPTION: how the AND Gate works
architecture func of andGate_timed is
begin
  process(CLK)
  begin
    if rising_edge(CLK) then
      F <= A and B;
    end if;
  end process;
end func;
```

Code C.2: Sequential AND-gate adapted from C.1

C.2 Shift register

```
-------------------------------------------------
-- 3-bit Shift-Register/Shifter
-- (ESD book figure 2.6)
-- by Weijun Zhang, 04/2001
--
-- reset is ignored according to the figure
-------------------------------------------------
library ieee ;
use ieee.std_logic_1164.all;
-------------------------------------------------
entity shift_reg is
port( I:     in std_logic;
  clock:     in std_logic;
  shift:     in std_logic;
  Q:     out std_logic
);
end shift_reg;
-------------------------------------------------
architecture behv of shift_reg is
    -- initialize the declared signal
    signal S: std_logic_vector(2 downto 0):="111";
begin
    process(I, clock, shift, S)
    begin
  -- everything happens upon the clock changing
  if clock'event and clock='1' then
      if shift = '1' then
    S <= I & S(2 downto 1);
      end if;
  end if;
    end process;
    -- concurrent assignment
    Q <= S(0);
end behv;
-------------------------------------------------
```

Code C.3: Shift register VHDL description downloaded from [15]

C.3 Uart tx line

```
1  -- This Source Code Form is subject to the terms of the Mozilla Public
   -- License, v. 2.0. If a copy of the MPL was not distributed with this
       file, You can obtain one at http://mozilla.org/MPL/2.0/.
3  -- Copyright (c) 2014-2015, Lars Asplund lars.anders.asplund@gmail.com
   library ieee;
5  use ieee.std_logic_1164.all;
   use ieee.numeric_std.all;
7  library vunit_lib;
   use vunit_lib.check_pkg.all;
9  use vunit_lib.log_pkg.all;

11 entity uart_tx is
     generic (
13     cycles_per_bit : natural := 434);
     port (
15     clk : in std_logic;
       -- Serial output bit
17     tx : out std_logic := '1';
       -- AXI stream for input bytes
19     tready : out std_logic := '0';
       tvalid : in std_Logic;
21     tdata : in std_logic_vector(7 downto 0));
   begin
23   -- pragma translate_off
     check_stable(clk, check_enabled, tvalid, tready, tdata, "tdata must be
       stable until tready is active");
25   check_stable(clk, check_enabled, tvalid, tready, tvalid, "tvalid must
      be active until tready is active");
     check_not_unknown(clk, check_enabled, tvalid, "tvalid must never be
      unknown");
27   check_not_unknown(clk, check_enabled, tready, "tready must never be
      unknown");
     check_not_unknown(clk, check_enabled, tx, "tx must never be unknown");
29   traffic_logger: process (clk) is
     begin
31     if tvalid = '1' and tready = '1' and rising_edge(clk) then
         debug("Sending " & to_string(to_integer(unsigned(tdata))));
33     end if;
     end process traffic_logger;
35   -- pragma translate_on
   end entity;
37
   architecture a of uart_tx is
39   signal tready_int : std_logic := '0';
```

```
begin
  main : process (clk)
    type state_t is (idle, sending);
    variable state : state_t := idle;
    variable cycles : natural range 0 to cycles_per_bit-1 := 0;
    variable data : std_logic_vector(9 downto 0);
    variable index : natural range 0 to data'length-1 := 0;
  begin
    if rising_edge(clk) then
      case state is
        when idle =>
          tx <= '1';
          if tvalid = '1' and tready_int = '1' then
            state := sending;
            cycles := 0; index := 0;
            data := '1' & tdata & '0';
          end if;
        when sending =>
          tx <= data(0);
          if cycles = cycles_per_bit - 1 then
            if index = data'length-1 then
              state := idle;
            else
              index := index + 1;
            end if;
            data := '0' & data(data'left downto 1);
            cycles := 0;
          else
            cycles := cycles + 1;
          end if;
      end case;
      if state = idle then
        tready_int <= '1';
      else
        tready_int <= '0';
      end if;
    end if;
  end process;

  tready <= tready_int;
end architecture;
```

Code C.4: UART transmit line VHDL description available in the vunit repository

C.4 SPI master

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity SPI_Master is  -- SPI-Modus 0: CPOL=0, CPHA=0
    Generic ( Quarz_Taktfrequenz : integer   := 50000000;  -- Hertz
              SPI_Taktfrequenz   : integer   :=  1000000;  -- Hertz
              Laenge             : integer   :=   8
            );
    Port ( TX_Data  : in  STD_LOGIC_VECTOR (Laenge-1 downto 0);
           RX_Data  : out STD_LOGIC_VECTOR (Laenge-1 downto 0);
           MOSI     : out STD_LOGIC;
           MISO     : in  STD_LOGIC;
           SCLK     : out STD_LOGIC;
           SS       : out STD_LOGIC;
           TX_Start : in  STD_LOGIC;
           TX_Done  : out STD_LOGIC;
           clk      : in  STD_LOGIC);
end SPI_Master;

architecture Behavioral of SPI_Master is
  signal   delay       : integer range 0 to (Quarz_Taktfrequenz/(2*
   SPI_Taktfrequenz));
  constant clock_delay : integer := (Quarz_Taktfrequenz/(2*
   SPI_Taktfrequenz))-1;
  type   spitx_states is (spi_stx,spi_txactive,spi_etx);
  signal spitxstate    : spitx_states := spi_stx;
  signal spiclk    : std_logic;
  signal spiclklast: std_logic;
  signal bitcounter    : integer range 0 to Laenge;
  signal tx_reg        : std_logic_vector(Laenge-1 downto 0) := (others
   =>'0');
  signal rx_reg        : std_logic_vector(Laenge-1 downto 0) := (others
   =>'0');
begin
   process begin
     wait until rising_edge(CLK);
     if(delay>0) then delay <= delay-1;
     else             delay <= clock_delay;
     end if;
     spiclklast <= spiclk;
     case spitxstate is
       when spi_stx =>
            SS          <= '1'; TX_Done     <= '0';
```

```
                bitcounter  <= Laenge; spiclk      <= '0';
42              if(TX_Start = '1') then
                   spitxstate <= spi_txactive;
44                 SS         <= '0';
                   delay      <= clock_delay;
46              end if;
          when spi_txactive =>
48              if (delay=0) then
                   spiclk <= not spiclk;
50                 if (bitcounter=0) then
                      spiclk     <= '0';
52                    spitxstate <= spi_etx;
                   end if;
54                 if(spiclk='1') then
                      bitcounter <= bitcounter-1;
56                 end if;
                end if;
58        when spi_etx =>
                SS      <= '1';
60              TX_Done <= '1';
                if(TX_Start = '0') then
62                spitxstate <= spi_stx;
                end if;
64     end case;
   end process;
66
    process begin
68     wait until rising_edge(CLK);
       if (spiclk='1' and  spiclklast='0') then -- SPI-Modus 0
70        rx_reg <= rx_reg(rx_reg'left-1 downto 0) & MISO;
       end if;
72   end process;

74   process begin
       wait until rising_edge(CLK);
76     if (spitxstate=spi_stx) then    -- Zurucksetzen, wenn SS inaktiv
          tx_reg <= TX_Data;
78     end if;
       if (spiclk='0' and  spiclklast='1') then -- SPI-Modus 0
80        tx_reg <= tx_reg(tx_reg'left-1 downto 0) & tx_reg(0);
       end if;
82   end process;
    SCLK    <= spiclk; MOSI    <= tx_reg(tx_reg'left); RX_Data <= rx_reg;
84 end Behavioral;
```

Code C.5: SPI master VHDL description downloaded from [16]

Appendix D

Testbench template

```
2 library ieee;
  use ieee.std_logic_1164.all;
4
  library vunit_lib;
6 context vunit_lib.vunit_context;

8 entity tb_entity_name_gen is
    generic(runner_cfg : runner_cfg_t);
10 end entity;

12 architecture entity_name_generated_testbench of tb_entity_name_gen is
    shared variable warning_logger : checker_t;
14   COMPONENT uut
       PORT();
16   END COMPONENT;
     --# Clock Cycles
18   --# Wait Times
     --# Relative Period
20   --# Helper Types
     --# Constants
22   --# Timing Signals
     --# Clock Signals
24   --# Input Signals
     --# Output Signals
26   --# Simulation Signals
  begin
28   dut : UUT PORT MAP();

30   checker_initiation : checker_init(warning, "", "output_directory/
```

```
    entity_name__test_name_messages.csv", level, verbose_csv, failure,
    ',', false);
  warning_logger_initiation : checker_init(warning_logger, warning, "",
    "output_directory/entity_name__test_name_result.csv", level,
    verbose_csv, failure, ',', false);

  main : process
    variable n : integer := 0;
    variable v : integer := 0;
    variable error_number     : integer := 1;
    --# Wait Variables

  begin
    test_runner_setup(runner, runner_cfg);

    while test_suite loop
      reset_checker_stat;
      n := 0;

      if run("test_name") then
        --# Loop start
        --# sig_n driver
        --# Extra Code For Waiting
          --# Stimulus rising edge
          --# Stimulus falling edge
          --# test checking
          n := n+1;
          --# Extra End If For waiting
        --# Loop end
        --# set endofsimulation
      end if;
    end loop;

    test_runner_cleanup(runner);
  end process;

  --# Clock driver

end architecture;
------------------------------------------------------------END
```

Code D.1: template.vhd