

# Automated testbench generation from digital timing diagrams

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 diagrams

University of Ghent



Seldeslachts Winand

May 20, 2016

# Abstract

# Acknowledgements

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Problem analysis</b>	<b>10</b>
<b>3</b>	<b>Functional analysis</b>	<b>13</b>
<b>4</b>	<b>Technical analysis</b>	<b>18</b>
4.1	Testbench creation . . . . .	19
4.1.1	Testbench specifications . . . . .	19
4.1.2	Conversion script . . . . .	21
4.1.3	Source file . . . . .	21
4.2	Wave trace analysis . . . . .	22
<b>5</b>	<b>Technical design</b>	<b>23</b>
5.1	Testbench creation . . . . .	23
5.1.1	Simple example . . . . .	23
5.1.2	Clocked designs . . . . .	28
5.2	Vectors and repetitions . . . . .	32
5.2.1	Vectors . . . . .	33
5.2.2	Application example . . . . .	38
5.3	Overview . . . . .	39
5.4	Wave trace comparison . . . . .	40
5.4.1	Practical approach . . . . .	42
5.4.2	Preparations . . . . .	45
5.4.3	WaveDrom result file . . . . .	49
<b>6</b>	<b>Conclusion</b>	<b>55</b>



# Chapter 1

## Introduction

In software and hardware development, verification and validation is the process of checking that 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 slow 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 an easy way to easily design tests and simulate them, while at the same time generating documentation. The simulation result is then automatically compared to the documentation. The operation of the resulting software is illustrated below.

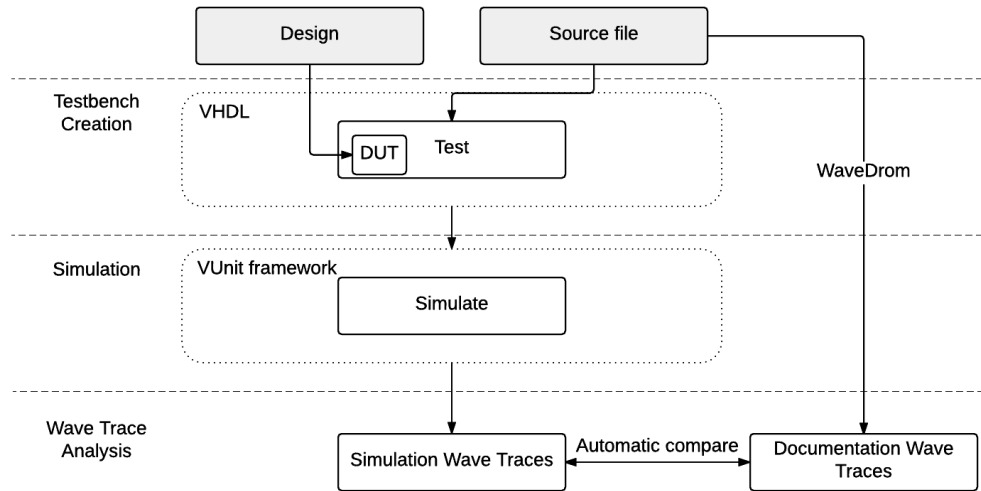


Figure 1.1: Operation overview of the new system

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

Whenever a hardware component is implemented in a hardware description language, its behavior is defined the design file. The behavior of the component is often documented visually by showing input and expected output wave traces. Wave traces are the visualisation of a signals values over time. An example for an AND-gate is shown below. Input signals are connected to a designs input and the output signals represent the output of the device.

The behavior is tested by attaching it to a so called testbench. Historically, it is also written in a HDL and describes the environment in which the design is to be tested. In this testbench one or more tests can be performed

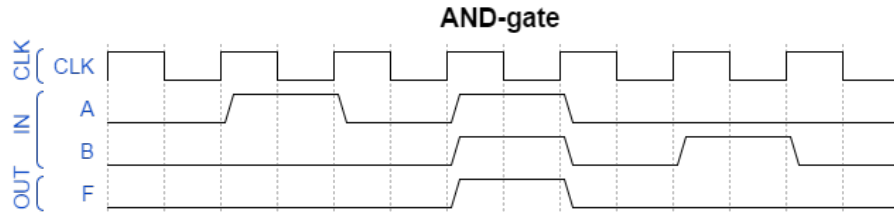


Figure 1.2: AND-gate wave trace example

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.

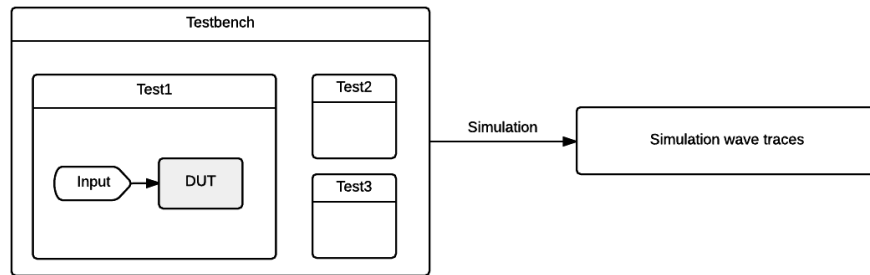


Figure 1.3: Hierarchy of a testbench

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).

## Maak Mij

Figure 1.4: Modelsim simulated wave traces

To verify a design, ideally, every possible input has to be simulated in the testbench and have its 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 verification cycle.

The complete verification cycle is shown in figure 1.5.

Each step can be optimized in their 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.

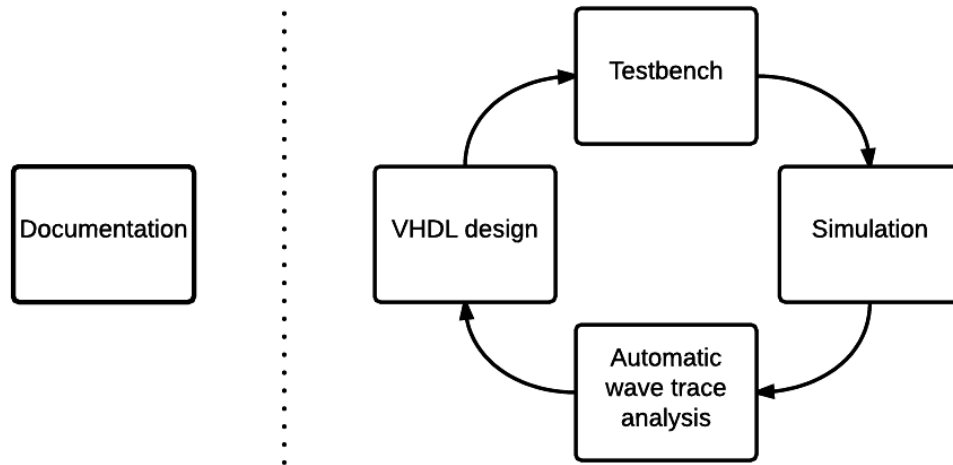


Figure 1.5: Traditional verification cycle

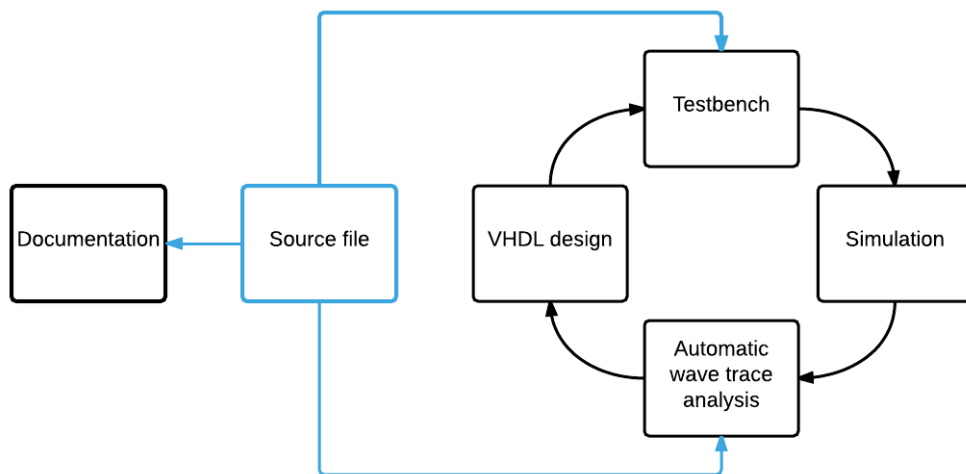


Figure 1.6: Improved verification cycle

# 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 visually 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 efficient 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 is time consuming and prone to human error. If this process could be automated, a lot of time and work could be saved. At the same time writing the documentation is a tedious task, where every function has to be explained. Documentation often includes example wave traces. As can be seen from image 1.2 and 1.4, which are documentation wave traces and simulated wave traces 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 essentially 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, eliminating the need to create testbenches manually, while still offering the same functionality.

In short, the system should be able to build a self checking testbench automatically based on the information in the documentation.

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. 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, which 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 easy to understand format. Building on the presumption that there is a way to build both documentation wave traces and testbenches from the same information, we can say that the documentation wave traces should be exactly the same as the simulated wave traces. Because of this wave trace analysis boils down to checking these two wave trace files for discrepancies and showing the difference in a clear way to the user.

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 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 §[REF PA]. To improve on this process the information provided in a source file is used. 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.

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 file based on standard WaveDrom [2] files. It 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.



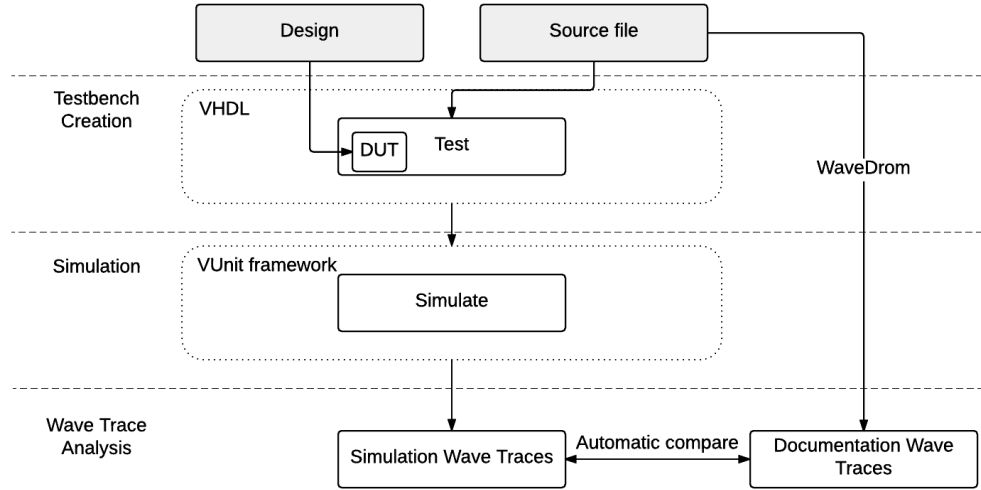


Figure 3.1: Operation overview of the new system

Then, the framework provided by VUnit is used to run the created testbenches through 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. The WaveDrom generated figure 3.2 illustrates this concept. This visual representation will be accompanied by an error message specifying the error as seen in log message 1.

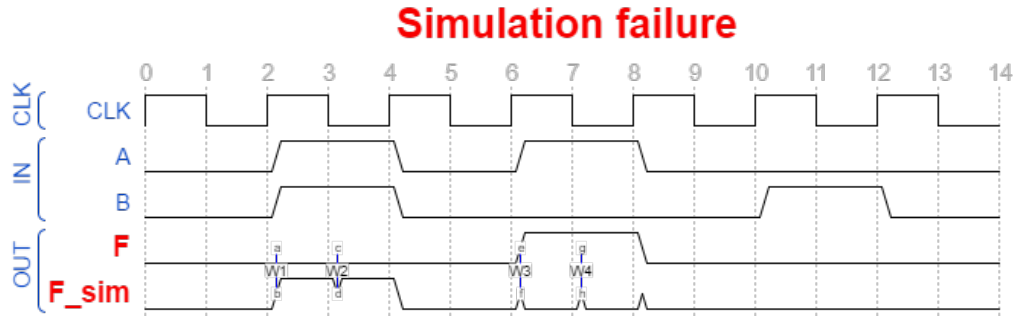


Figure 3.2: Simulation failure example

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.

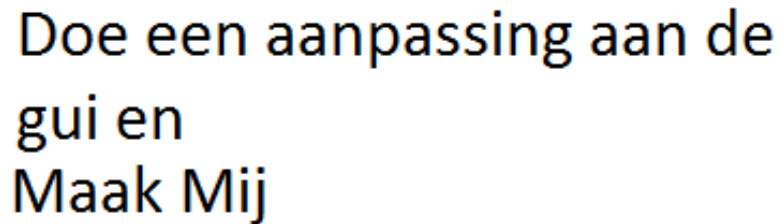
The image shows a window with a title bar. The text inside the window is arranged in four lines: 'Doe een aanpassing aan de', 'gui en', 'Maak Mij'. The text is in a dark blue, sans-serif font. The window appears to be a standard graphical user interface element.

Figure 3.3: Look of the user interface

The gui allows the user to open the input folder where WaveDrom input files should be added, the src 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 json file in the source folder, simulates the result, verifies it and stores the result 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
- 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 way of compared to what it used to be - (create testbench,) simulate, compare

simulation wave traces manually. 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 other frameworks such as 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.

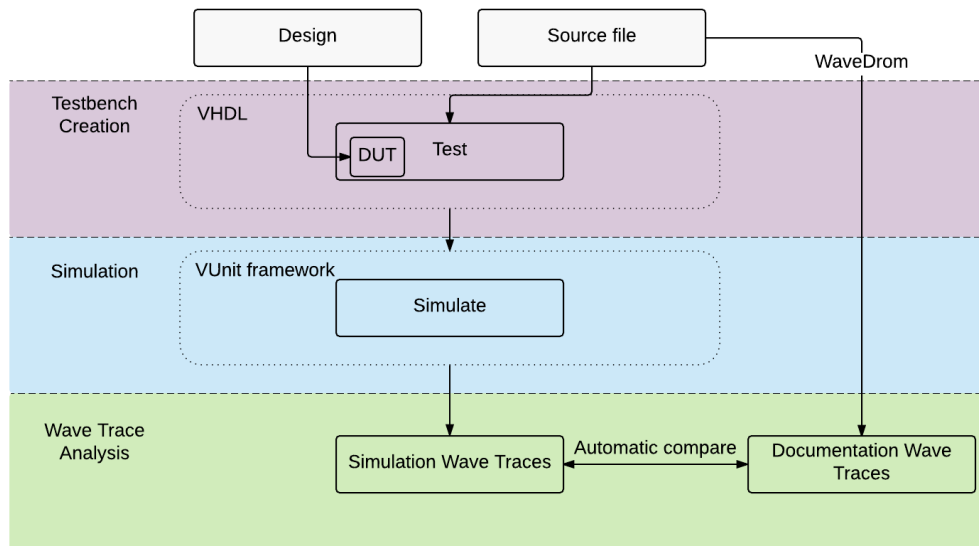


Figure 4.1: Color coded overview of the program operation

## 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. This part corresponds to the purple part.

### 4.1.1 Testbench specifications

Before the way testbenches should be constructed can be defined, the specifications for the testbenches have to be determined. This depends on which framework is 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 different programming languages.

Some of the most known frameworks, their supported languages and their most important features are listed below. 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

- AANVULLEN

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.

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

Code 4.1: Main test loop for a VUnit testbench

Multiple 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 test can be started by a python command. These test 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 test 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 relevant information 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 optimisation 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 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 files in this report.

WaveJSON [?] 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 test-bench. 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 in 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 files in which the 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 description of the development of the software. Every new element is introduced at the hand of an increasingly complicated example input file.

#### 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 program conversion script can be created based on a simple combinatorial design.

The example this draft was based on was an AND gate. The AND-gate design can be found in ??.

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.

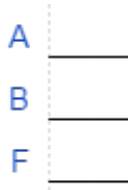


Figure 5.1: Wave traces corresponding to description in code 5.1

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:

```
1 { "name": "andGate", "test" : "andgate00",
2   "description": "Input A = '0' & B = '0' should produce a
   low output",
3   "signal": [
4     ["IN",
5       { "name": "A", "wave": "0", "type": "std_logic" },
6       { "name": "B", "wave": "0", "type": "std_logic" } ],
7     ["OUT",
8       { "name": "F", "wave": "0", "type": "std_logic" } ]
9   ] }
```

---

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

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

### 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 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.

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

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

Code 5.3: Generating signal declarations in Python

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

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

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

7        #example: "sig_example <= '1';"
        stimulus += "sig_" + signal["name"] + " <= '" + value
        + "';\n"
9    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.

```
1 def generate_port_map(json_signal_set):
    port_map = "PORT MAP(\n"
2     first = True
    for signal_type in json_signal_set:          # IN and OUT.
3         if len(signal_type) > 0:              # JSON allows empty
            elements
            for signal in signal_type:
4                 if len(signal) > 0:          # JSON allows empty
                    elements
                    if first:
5                        #example: "example => sig_example"
                        new_map = signal["name"] + " => sig_" + signal[
6                            "name"]
7                        first = False
                    else:
8                        new_map = ",\n" + signal["name"] + " => sig_" +
9                            signal["name"]
10                       port_map += new_map
11 port_map += " );"
12 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 port map is added to the list. The final list 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 organise 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 recognised by the program and replaced 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 ??

The space for input and output signals is marked by the keywords '`--# Input Signals`' and '`--# Output Signals`' respectively. Every 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 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 us to create more extensive tests. The AND-gate VHDL design can be found in appendix ?? or at [9].

The WaveDrom source file for testing all possible inputs and checking all corresponding outputs for an AND-gate is shown in code [ADD REF]. The generated wave traces are shown below.

---

```
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",
6         "period": "2", "clock_period": "20" }],
7     ["IN",
8       { "name": "A", "wave": "01010..", "type": "std_logic", "
9         period": "2" },
10      { "name": "B", "wave": "0.....1.0.1.0.", "type": "
11        std_logic" }],
12     ["OUT",
13       { "name": "F", "wave": "0.....1.0.....", "type": "
14         std_logic" }]]
15 }
```

---

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

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

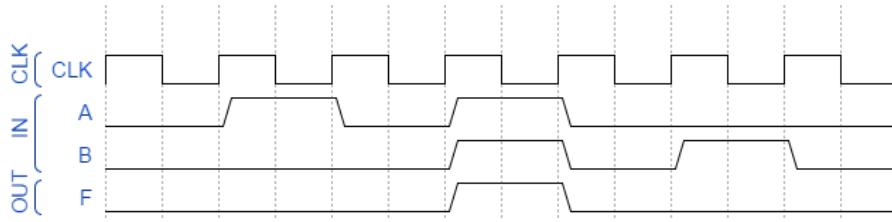


Figure 5.2: Wave traces generated by WaveDrom based on the source file in code 5.6

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 two things. A conversion of the JSON signal value representation to a VHDL representation and knowing the logic type of the signal concerned. Converting the JSON signal value is for the most part simply expanding the "value" string and converting the separate



characters (e.g. the JSON wave equal to "0.1." will be converted into ('0', '0', '1', '1')). As mentioned in the beginning of this paragraph the signal type must also be known, this is because an array type can only be defined if the type contained in it and its length 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.

```
1 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 length `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. 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 ??:

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

Code 5.8: Definition of a `std_logic` array

We can assume the test will be defined for two clock cycles and the `clock_cycle` constant is equal to 2.

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 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 is to assign the next value and compare the output signals each new clock cycle. This is illustrated in code 5.10 in the next paragraph.

### 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.

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

Code 5.9: Definition of a std\_logic array

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

Code 5.10: Signal driving in clocked designs

The clock driving process is not design dependant and is simply added to the testbench 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 will be driven on the rising edge of the internal clock and every output signal will be checked on the falling edge of the internal signal. The clock changes twice every internal clock period and has to be driven on the falling edge as well as the rising edge. For the timed AND-gate example the driving and checking process of the signals is shown in code 5.11.

```
1 while (n <= clock_cycles -1) loop
    wait until rising_edge(internal_clock);
3    -- CLK has twice as many values as other signals
    sig_CLK <= sig_CLK_values(2*n);
5
    sig_A <= sig_A_values(n);
7    sig_B <= sig_B_values(n);
```

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. For example the actual clock can run twice as slow as the internal clock by setting the "period" field to 2. This allows output signals to change on the falling edge of a high-to-low clock signal, which was not possible when all signals were driven on the rising edge of the actual clock signal. The internal clock period will define the length of one time step and will rise and fall within one time step. This way each signal is driven and checked every time step regardless 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 ??.

### 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",
6         "period": "2", "clock_period": "20" } ],
7     ["IN",
8       { "name": "tvalid", "wave": "0.1..0.....", "type":
9         "std_logic" },
10      { "name": "tdata", "wave": "=.=.=.....", "data":
11        [ "0", "249", "0" ], "type": "std_logic_vector", "
12        vector_size": "8" } ],
13     ["OUT",
14       { "name": "tx", "wave": "1....0.....", "type": "
15         std_logic" },
16       { "name": "tready", "wave": "01.0.....", "type":
17         "std_logic" } ]
18   ] }

```

---

Code 5.12: Source file for creating the first transmission test for the UART design in ??

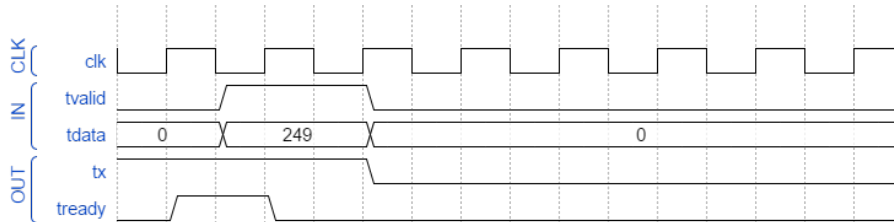


Figure 5.3: Wave traces generated by WaveDrom from code 5.12

The image above 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, it will now take two. This allows the tvalid signal to be 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, in stead 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.

---

```

1  {"name": "uart_tx", "test" : "uart_send_1_byte",
2  "description": "Send one byte with a parallel to serial
   uart.",
3  "signal": [
4  ["CLK",
5   {"name": "clk", "wave": "n.....|", "type": "std_logic",
    "period": "2", "clock_period": "20", "loop_times" : ["
    10*434"]}],
6  ["IN",
7   {"name": "tvalid", "wave": "0.1..0.....", "type":
    "std_logic"},
8   {"name": "tdata", "wave": "=.=..=.....x.", "data":
    ["0", "249", "0"], "type": "std_logic_vector", "
    vector_size" : "8"}],
9  ["OUT",
10   {"name": "tx", "wave": "1....0.....x.", "type": "
    std_logic"},
11   {"name": "tready", "wave": "01.0.....x.", "type":
    "std_logic"}]
12 ]}

```

---

Code 5.13: Source file for creating a second transmission test for the UART design in ??

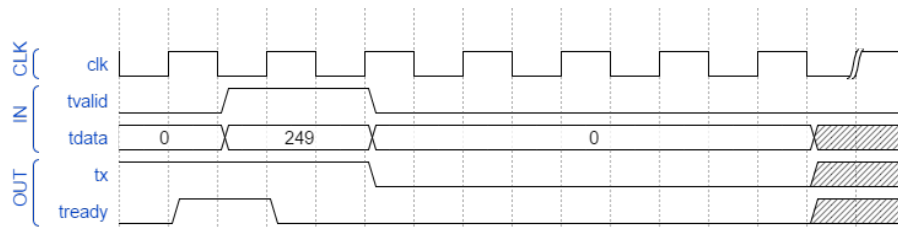


Figure 5.4: Wave traces generated by WaveDrom from code 5.13

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

$$8 + 10 * 434 = 4348 \quad (5.1)$$

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*434", "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", "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 ??



Figure 5.5: 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 ?? in more detail.

### 5.3 Overview

The full code can be found online [10]. An overview of the code functionality is given in image 5.6.

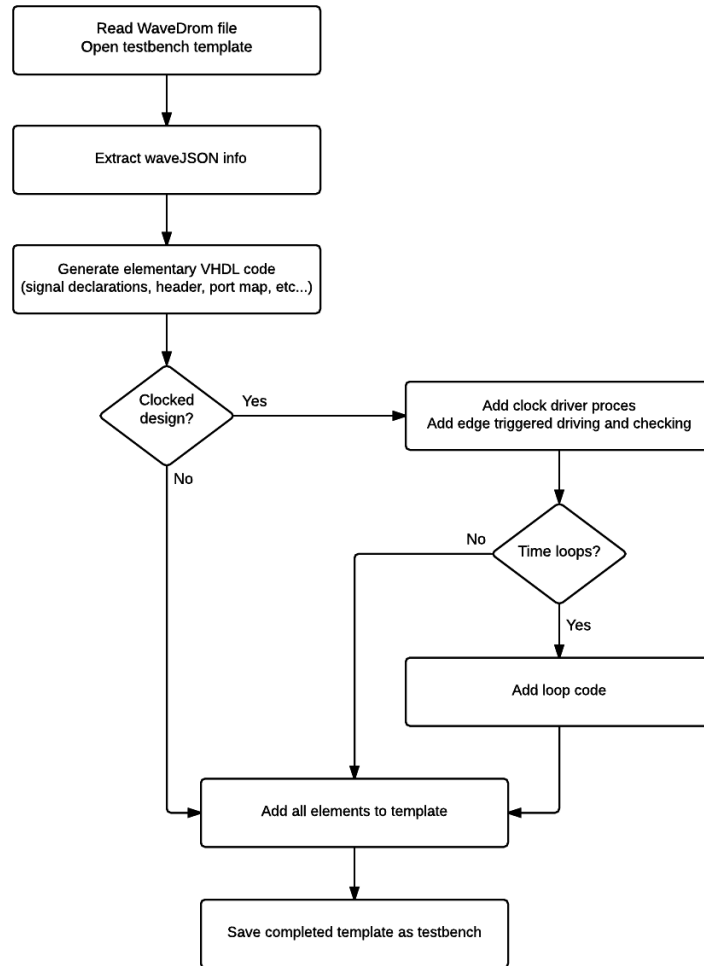


Figure 5.6: Overview of the testbench creation code

The code is run for every WaveDrom file in the resource folder (see appendix ?? user guide). The method used in this program can be compared to building a prefabricated house. First the walls, cellar, floors and roof are fabri-

cated 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. They include signal declarations, uut declaration and uut port map. A full list can be found in §???. 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 if 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 that 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

Currently 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. 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.

### 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.

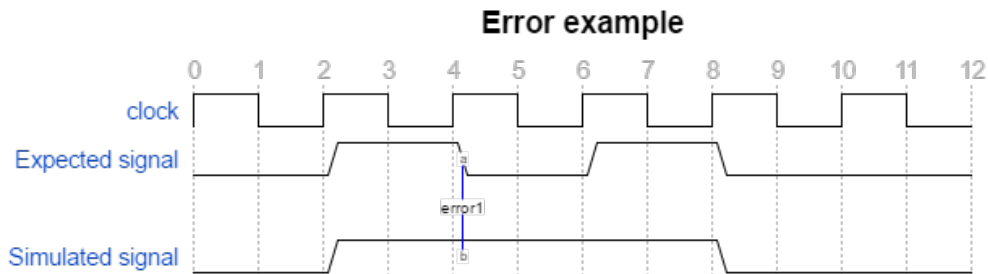


Figure 5.7: Ideal error marking

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 now 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

$$x = n / \text{clock\_period} \quad (5.2)$$

and clock\_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 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.

### 5.4.1 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 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 there is already a way to link the WaveDrom generated wave trace file to the one generated by the simulation all that is left is to add the error messages logged during simulation. 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
2  check(sig_example = sig_example_values(n),
  "This check failed. Expected example = " &
4  std_logic'image(sig_example_values(n)) & ",
  got example = " & std_logic'image(sig_example) &
6  " 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 ?? The code below initialises the universal checker and is executed on testbench start.

```
1 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 to be added to. This way the log message from the check in code 5.15 is added to a .csv file in a subfolder named ".warning\_log/" and contains the entity name of the testbench it is derived from. The message 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."

Which directly links the actual error to a specific point in time. From the current system it should be possible to derive the system proposed as the ideal solution in §5.4.

### 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 §??.

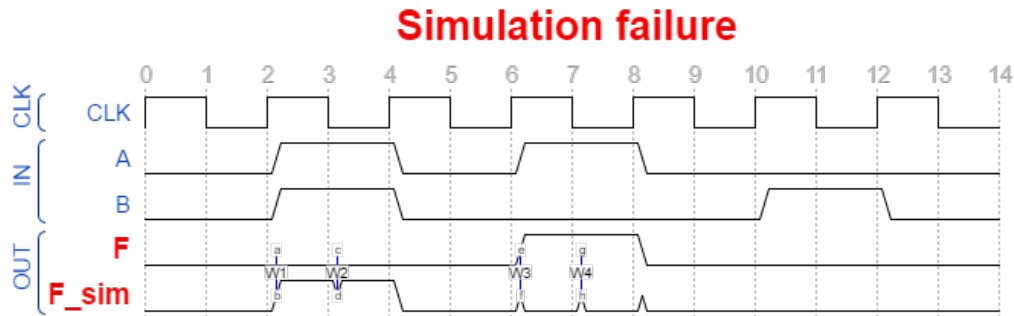


Figure 5.8: Layout proposal for result files

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.

### 5.4.2 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 code ??.

---

```

1  { "name": "andGate_timed", "test" : "andgate_failing",
2  "description": "a full AND-gate test designed to fail", "
    signal": [
3    ["CLK",
4      { "name": "CLK", "wave": "p.....", "type": "std_logic",
        "period": "2", "clock_period": "20" } ],
5    ["IN",
6      { "name": "A", "wave": "01010..", "type": "std_logic", "
        period": "2" },
7      { "name": "B", "wave": "0.1.0.....1.0.", "type": "
        std_logic" } ],
8    ["OUT",
9      { "name": "F", "wave": "0.....1.0.....", "type": "
        std_logic", "node": "..ac..eg" },
10     { "name": "F_sim", "wave": "0.110.000.....", "type": "
        std_logic", "node": "..bd..fh" } ]
11 ],
12 "head": { "text": ["tspan", { "class": "error h3" }, "
    Simulation failure "], "tick": 0 },
13 "edge": ["a-b W1", "c-d W2", "e-f W3", "g-h W4"]
14 }

```

---

Code 5.16: Source file for the proposed layout in fig 5.8

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

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

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 5.4.1).



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.

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

Code 5.17: Custom checker declaration

```
1 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:

```
1 if sig_example_values(n) /= 'X' then  
  check(sig_example = sig_example_values(n),  
3    integer'image(error_number) & " " -- original message is  
    added here);  
  check(warning_logger, sig_example = sig_example_values(n)  
5    ,  
    integer'image(error_number) & " " -- new log message is  
    added here, line_num => error_number);  
  if sig_example /= sig_example_values(n) then  
7    error_number := error_number + 1;  
  end if;  
9 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 2: 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 3: Log messages in the "andgate\_failing\_result.csv" file

Now all information can be 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.

If the log file is empty and the test has succeeded, all that has to be done is add a header with an internal clock counter and a title indicating success. If the test failed and the log file is not empty the title should indicate failure and the errors should be shown as in figure 5.8.

### 5.4.3 WaveDrom result file

As stated in §?? 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 "/result"). The flow diagram below shows the function of the software that generates the compare files.

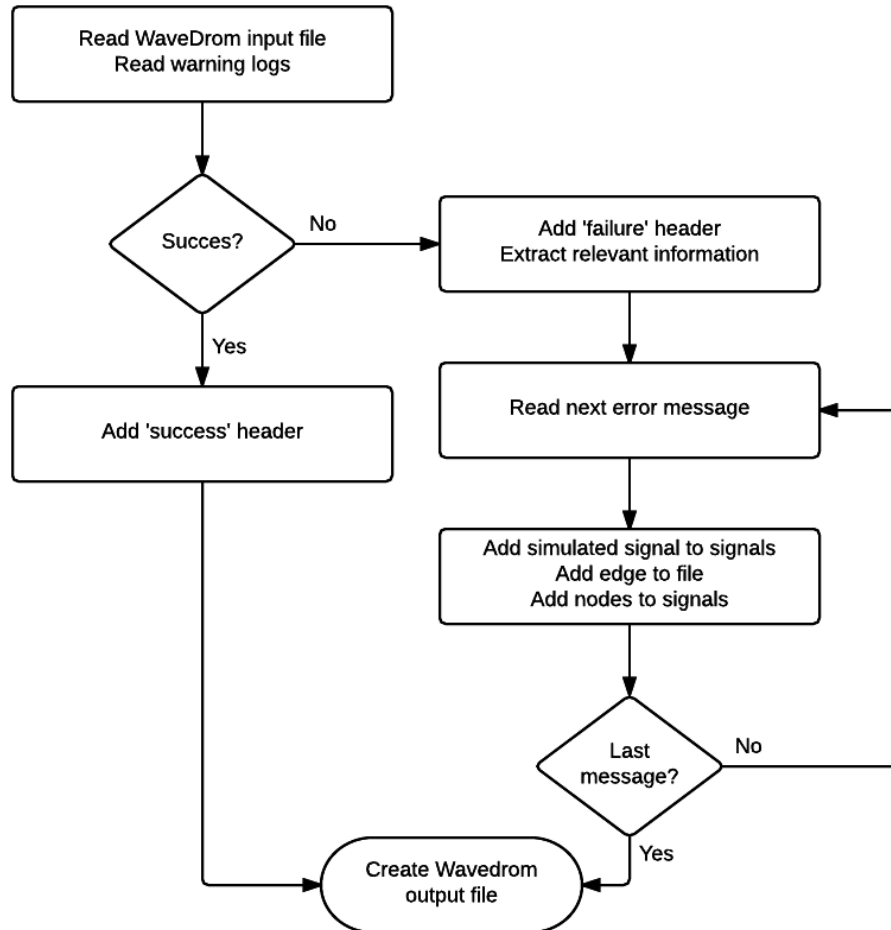


Figure 5.9: Overview of the wave trace analysis code

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 then add a header and create the output file. In case there are messages logged however, the simulation failed and the process becomes more complicated.

In case of failure a corresponding header is added containing a title and the time steps. 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:

---

```
1 { "name": "andGate_timed", "test" : "andgate_failing",
2   "description": "a full AND-gate test designed to fail",
3   "signal": [
4     ["CLK",
5      { "name": "CLK", "wave": "p.....", "type": "std_logic",
6        "period": "2" }],
7     ["IN",
8      { "name": "A", "wave": "0.1.0.1.0.....", "type": "std_logic" },
9      { "name": "B", "wave": "0.1.0.....1.0.", "type": "std_logic" }],
10    ["OUT",
11     { "name": "F", "wave": "0.....1.0.....", "type": "std_logic" }]]
12 ] }
```

---

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

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'. The wave traces for this signal are exactly equal to the wave traces of 'F', which means that it holds the same errors. These errors 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.

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

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

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

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

Log message 4: Log messages in the "andgate\_failing\_result.csv" file

The 'F\_sim' signal is now added to the output data.

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 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:

---

```
1  { "name": "andGate_timed", "test" : "andgate_full", "
    description": "Test all possible inputs for an AND-gate
```

---

```

    , "signal": [
2  ["CLK",
3   {"name": "CLK", "wave": "p.....", "type": "std_logic",
    "period": "2", "clock_period": "20"}],
4  ["IN",
5   {"name": "A", "wave": "01010..", "type": "std_logic", "
    period": "2"},
6   {"name": "B", "wave": "0.1.0.....1.0.", "type": "
    std_logic"}],
7  ["OUT",
8   {"name": "F", "wave": "0.....1.0.....", "type": "
    std_logic", "node": "..a"},
9   {"name": "F_sim", "wave": "0.1.0.....", "type": "
    std_logic", "node": "..b"}]
10 ], "head": {"text": ["tspan", {"class": "error h3"}, "
    Simulation failure "],
11 "tick": 0
12 }, "edge": ["a-b W1"]
13 }

```

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

Which yields:

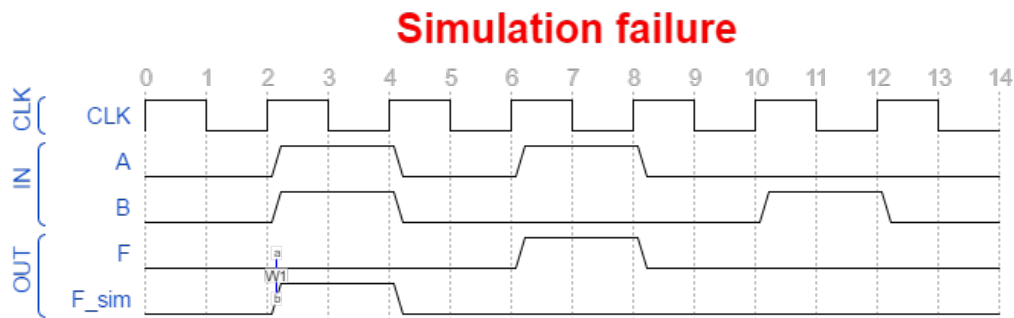


Figure 5.10: Wave traces generated by WaveDrom from code 5.21

After this, the second message is read and processed the same way. The only difference is that 'F\_sim' already exists. In this case simulation signal is altered rather than created and the node field is extended.

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

---

```

1  {"name": "andGate_timed", "test" : "andgate_failing",
2  "description": "a full AND-gate test designed to fail", "
   signal": [
3  ["CLK",
4  {"name": "CLK", "wave": "p.....", "type": "std_logic", "
   period": "2", "clock_period": "20"}]},
5  ["IN",
6  {"name": "A", "wave": "01010..", "type": "std_logic", "
   period": "2"}],
7  {"name": "B", "wave": "0.1.0.....1.0.", "type": "std_logic"
   }]},
8  ["OUT",
9  {"name": "F", "wave": "0.....1.0.....", "type": "
   std_logic", "node": "..ac..eg"}],
10 {"name": "F_sim", "wave": "0.110.000.....", "type": "
   std_logic", "node": "..bd..fh"}]}
11 ],
12 "head": {"text": ["tspan", {"class": "error h3"}, "
   Simulation failure "], "tick": 0},
13 "edge": ["a-b W1", "c-d W2", "e-f W3", "g-h W4"]
14 }

```

---

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

This yields figure 5.11.

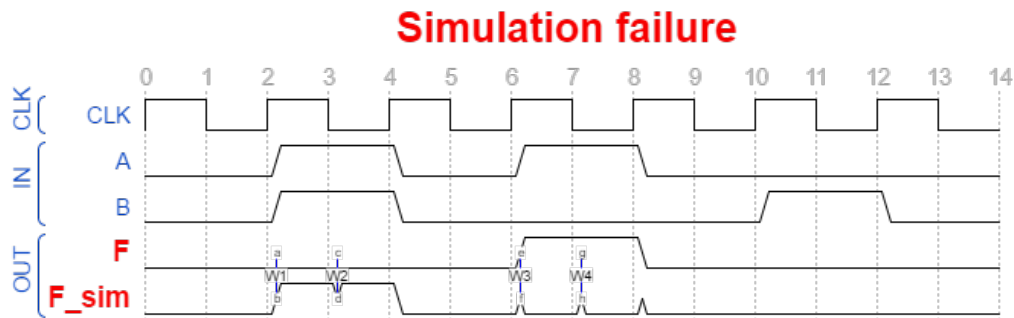


Figure 5.11: Wave traces generated by WaveDrom from code 5.22



Which is exactly the same as the goal image in image 5.8.

## Chapter 6

## Conclusion

# Appendix A

## example

# Bibliography

- [1] “Software verification and validation.” [Online]. Available: [en.wikipedia.org/wiki/Software\\_verification\\_and\\_validation](https://en.wikipedia.org/wiki/Software_verification_and_validation)
- [2] “Wavedrom homepage.” [Online]. Available: [wavedrom.com](http://wavedrom.com)
- [3] “Vunit home page.” [Online]. Available: [vunit.github.io](http://vunit.github.io)
- [4] “Cocotb documentation.” [Online]. Available: [cocotb.readthedocs.io/](http://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](http://vunit.github.io/documentation)
- [9] “Online examples for this project.” [Online]. Available: <https://github.com/WinandS/Thesis/tree/master/examples>
- [10] “Project source repository.” [Online]. Available: [github.ugent.be/wseidesl/Thesis](https://github.com/wseidesl/Thesis)