Building a better VHDL testing environment

Joren Guillaume

Supervisors: Prof. Luc Colman, dhr. Hendrik Eeckhaut (Sigasi)
Counsellor: dhr. Lieven Lemingre (Sigasi)

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

Department of Industrial Technology and Construction
Chairman: Prof. Marc Vanhaelst
Faculty of Engineering and Architecture
Academic year 2014-2015

Usage restrictions

Preface

Dankwoord en zo verder.

Abstract

Contents

List of Figures

List of Acronyms

Part I
Problem and background

1 Problem

Revisie nodig: belangrijkste pagina Developing digital hardware VHSIC Hardware Description Language (VHDL) is, like any code, prone to errors, either by the developer or by wrong product specifications. To ensure errors are weeded out before the more expensive roll-out or production begins, the code must be subjected to rigorous testing. *Currently, large and impractical tests are needed to fully test a product.*

Developing digital hardware VHDL is, like any code, prone to errors, either by the developer or by wrong or bad *synoniem* product specifications. To ensure errors are weeded out before the more expensive production begins*Zie commentaar*, the code must be subjected to rigorous testing. *Currently, large and impractical tests are needed to fully test a product. productie != einddoel alle VHDL -> FPGAs en FPGA verdelingen*

Because testing is such a time consuming process, finding errors often results in severe delays due to the need to both correct the error and test for others. Therefore it is in the best interests of both testing- and software engineers to write testbenches with maximal coverage, optimal readability and minimum time spent. It is also important to find and correct errors with minimal delay and maximal efficiency. This entire process should affect the least amount of code possible in order to minimize time spent retesting and modifying the code. *Betere samenvatting + meer gevolgen*

In this thesis, the objective is to explore the possibility of creating an operating system independent framework. This framework should allow users to quickly and consistently create, modify, execute and evaluate testbenches. To accomplish all of this, a number of industry standard tools will be incorporated around a central Python script. This combination should ensure timely and automated building, testing and test report generation.

2 Introduction

2.1 Hardware Description Languages

Meer bronvermelding

A Hardware Description Language (HDL) can be used to describe digital electronics, i.e. hardware, in different levels. The level that uses certain blocks of logic gates to describe more complex behaviour is called the Register Transfer Level (RTL). Some blocks are standard implementations that have been widely used and are nearly fully optimized, such as memories, flip-flops and clocks. VHDL is such a language, having been developed in the eighties of the twentieth century, originally to have a uniform description of hardware brought in by external vendors to the U.S. department of defence. It was quickly realized that simulation was possible with a good description and the language evolved to be used as such. The final step was to create tools that could not only simulate, but also synthesize (i.e. create actual hardware layouts) from these descriptions. An RTL flip-flop implementation written in VHDL is shown here:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY dff IS
    PORT(d   : IN  std_logic;
         clk : IN  std_logic;
         q   : OUT std_logic;
END dff;

ARCHITECTURE Behavioural OF dff IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF rising_edge(clk) THEN
            q <= d;
        END IF;
    END PROCESS;
END Behavioural;
```

The Institute of Electrical and Electronics Engineers (IEEE) 1164 library provides a number of extensions on the original VHDL IEEE 1076 specification that allow a more realistic simulation and description of hardware behaviour. An *entity* defines the inputs and outputs of a certain building block, in this case the D Flip-Flop (DFF). The *architecture*, in this case Behavioural, takes the description of an entity and assigns a real implementation to it. In this architecture, we have one process and it is *sequential*, meaning that every update in the process follows an update of the *clk*, the clock signal. The d stands for delay, and it simply puts on its output, q, that which was on the input, d, one clock period earlier. All *processes* are executed in parallel, this does not mean that all are triggered at the same time, nor do they take equally as long to finish, but it means that any process can be executed alongside any other process. In this case there is only one (nameless) process that describes the entire behaviour of the flip-flop. The process waits for the rising edge of the clock, which is a transition from zero to one, after which it schedules the value of d to be put on q when the next rising edge appears.

Hierarchisch testing: figuur, uitleg
Wat is een testbench bvb, hoe werkt dit? niet te veel stappen overslaan

This is a basic example of an entity, an architecture and a process. Before the code can be put to use in a working environment, it needs to be tested first. This is done through the use of *testbenches*. [1] Testbenches are made up of code that takes a certain building block, the Unit Under Test (UUT) or Device Under Test (DUT). The testbench then puts a certain sequence of values on the inputs and monitors the outputs. Applying this to the example listed earlier, the testbench would contain a process with a clock, signals coupled with the ones in the entity, some stimuli and *wait* statements. The signals are linked to the DFF, which is now the UUT. Then the clock starts ticking and the input d is made '0' or '1' every now and then. All that is left is to assure the output q is always '0' and '1' exactly one clock cycle later. The full code is listed in appendix A.1.

If the device performs normally, the received output sequence should match an expected output sequence. In these testbenches it is good practice to observe how well a device performs if its inputs behave outside of the normal mode of operation. When all of these tests have finished and the output performs as expected, the device is ready to be put into production or further down the developmental process.

The higher level components, such as a registry, employ a number of the lower level ones to create more complex logic. The flip-flop could be used in certain numbers to build a *register*, a collection of ones and zeroes (henceforth referred to as *bits*) that is used to (temporarily) store these values. The register could then be used alongside combinational logic to build an even bigger entity. The idea here is that small building blocks can be combined to produce vast and complex circuits that are nearly impossible to describe in one go. Adding all these layers together also creates a lot of room for error, and having a multi-level design makes it challenging to pinpoint the exact level and location of any errors.

If a device is not tested properly and faults propagate throughout development, they can be very expensive to correct, especially if it is brought into production, where a single photomask, used to "print" part of the layout, can easily cost $100,000 (€81,000 at time of writing)[2]. Therefore a large portion of time is spent writing and executing tests.

Relevantere figuur, betere design flow

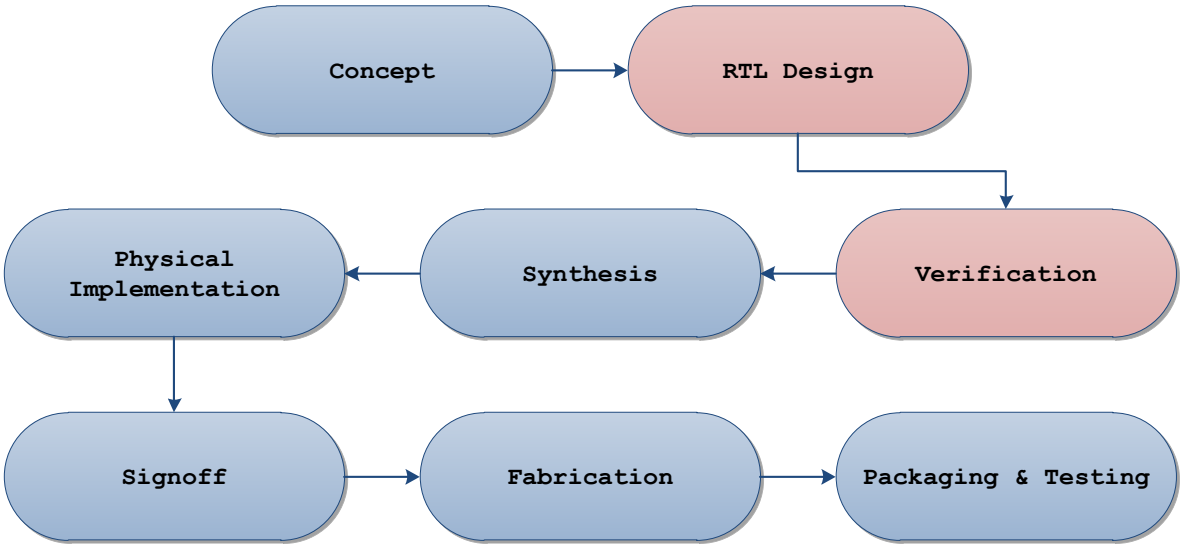

Figure 1: Typical ASIC design flow

3 Current industry practices

Klassieke testbench nog uitleggen

As mentioned before, a number of practices exist to improve speed and quality of testing and coding. The bigger part of these practices are applied in the software development industry, where this is quite literally their entire business. Moving to HDLs, it stands to reason that code meant for synthesis may not be able to follow all of these best practices. However, the industry has formed several practices and *methodologies* to try and create a uniform verification process. The most well-known will be discussed in some detail below.

3.1 Assertions

Assertions are the standard practice of verification. An assert in VHDL is very simple: check whether some boolean condition is true. If true, do nothing, if false, return the error message and throw an error of a certain level, as in the following example:

```
assert (not Q) report "Unexpected output value" severity failure;
```

In this example, an error is thrown of the severity *failure*, which ends the simulation, if the value on the output Q (see section 2.1)) isn't *logically true*. This means the output has to have a value of *high* or *1*. The severity of the assertion can be in the range of notice to failure, and the simulator might be set to respond or stop only to a certain level of severity. Doing this for the right values at intervals gives the developer a quick overview of whether everything went according to plan. After all, if things went wrong, the assertion should have thrown an error here or there.

3.2 Generic testbench

The generic testbench operates as mentioned in section 2.1. It assigns every input and output on the DUT their counterpart in the testbench and contains a number of processes with stimuli. Standard procedure is:

1. Wait for some clock periods to 'ready' the design
2. Apply stimuli to the inputs
3. Wait for the appropriate number of clock cycles
4. Use asserts to check whether the outputs have the right values
5. Repeat steps 2 through 4 until satisfied
6. End with an infinite 'wait' to suspend the process

Creating this kind of testbench for a large, hierarchical project would certainly become lengthy and unclear. To counter these disadvantages, a number of practices were created to keep control over what and how designs are tested. The more used of these practices are explained below.

3.3 Coverage

Coverage is a generic term that is used to describe how fully a design has been tested on one aspect or another. There exist many tools for different coverage analyses, but in this section we will focus only on the types of coverage.

3.3.1 Code Coverage

In development, Code Coverage (CC) is a type of measurement to indicate how well the source code has been tested. With the use of a coverage report, unused blocks of code can be uncovered, these blocks might indicate unnecessary code or a bug. Imagine a Finite State Machine (FSM) with an unused reset state, this might indicate that the reset isn't functioning properly or there are no tests covering the reset. CC does not, however, provide any real functional analysis. It does not indicate any missing lines of code nor does it tell you whether the inputs and outputs behave properly.

3.3.2 Functional Coverage

Controleren
Functional Coverage (FC) is the practice of measuring whether the UUT meets with certain specifications at specific times during the testing process. These specifications are created by the developer and are used to check whether the design performs as expected. Good practice is to include corner cases, cases that cover very rare occurrences and so on. That way, the device is sure to be in working condition even under unexpected circumstances.

3.4 Verification

3.4.1 Constrained Random Verification

Revision! Voorbeelden, niet noodzakelijk VHDL, bronvermelding (algemeen)
Resultaat en hoe controleren bij random input
Constrained Random Verification (CRV) is an industry practice where one or more inputs are generated randomly, within certain bounds or *constraints*. This practice was brought into use after designs grew too large for Directed Testing (DT) to support. DT has verification engineers write

out very specific things they want to test, for instance, a reset pulse to verify the reset working correctly. CRV opposes this with the idea that for all behaviour to be tested properly in large designs, the amount of time spent writing and executing tests would simply become too great. It proposes a solution where inputs are generated randomly, within certain bounds, but in a sufficiently large quantity to have implicitly covered all scenarios. It is important to note that in DT, expected behaviour is directly tested, but in CRV it is likely to be the unexpected behaviour that gets tested too. This solved the long standing problem of testing any behaviour, including the unexpected.

3.4.2 Formal Verification

On top of the aforementioned, there are several more practices that have unique ways of verifying the properties of a design, but aren't used sufficiently to merit full detailing. Formal verification is such a practice, where the core idea is to mathematically prove the design from its specifications. The upside is that the design is completely verified, however, it is out of use because proving large designs is not only tedious but takes up large amounts of man-hours.

3.5 Simulation tools

As mentioned in section 2, HDLs are used for developing hardware, and need to be tested and build as such. Like any other programming language, they need a dedicated compiler to fault-check the code and build the binaries. Unlike other programming languages they need a simulator in order to verify the builds. There exist many compilers and simulators, almost none are exclusive for VHDL, almost all are dual-language and support some form of Verilog, a different HDL. However, many of them refuse to support the latest additions to the VHDL language specification, even the 2002 additions are hard to be found. [3],[4],[5],[6]

3.5.1 ModelSim

ModelSim is the simulator that was investigated for several reasons. First, a free student edition was available. Considering licenses outside of school can easily cost upward of $25,000 (€20,250 at time of writing) this made it ideal for any thesis or student related work. Proof of this is the extensive use of ModelSim in the courses related to HDL development. Second, ModelSim supports all versions of VHDL, which is the HDL we are processing. Even in 2014, only one other tool was found to support all of VHDL-2008, which is Aldec's *Active-HDL*. And last but not least, ModelSim is one of the industry's most used simulators, enabling a pool of experience to be consulted.[7]

Figuurtje of vergelijking tussen simulators?

4 Software development practices

During the making of this thesis, a number of approaches were investigated and some were put to practical use. In this chapter, the more useful and tried of the aforementioned are discussed in detail.

4.1 Test Driven Development

Test Driven Development (TDD) is a proven development technique that has regained traction in the past decade, primarily through the efforts of Kent Beck [8]. This practice has proven to increase test coverage [9], decrease defect density [10] as well as improve code quality [10, 11]. The technique

focuses on tests being created before the actual code. It is important to make certain distinctions before going more in depth on the used methods. The developing community has a great many practices, each with their own names and methods, and hardly none are mutually exclusive.

4.1.1 Unit Testing

To understand TDD, a basic knowledge of *Unit Testing* is required. In software testing, a unit test is a test designed to check a single unit of code. Ideally, this unit is the smallest piece in which the code can be divided. A unit test should always test only a single entity, and only one aspect of that entity's behaviour. This division in units has a number of benefits, one of the most important being that code is exceedingly easy to maintain. Furthermore, the division of the code makes changes, when needed, fast to be carried out and ensures that only the modified code needs to re-tested.

4.1.2 Test First Development

Herschrijven Another main component of TDD is Test First Development (TFD), a technique that has a developer writing tests first, before any code has been written. This method makes the developer think on what the code has to achieve, rather than what the specific implementation has to be. A key feature of a new test is that it has to fail during its first run. If not, the test is obsolete seeing as the functionality it tests has already been implemented. After being run for the first time (and failing), the developer implements just enough code to get the test to pass. Once the test succeeds, it is time for a new test.

4.1.3 Refactoring

The third pillar of TDD is refactoring. After the tests succeed, it is necessary to clean up the code. A well-done TFD implementation uses the bare amount of code needed to make the test pass, but this is of course usually not the best code possible. Refactoring means that you take existing code and modify only the code itself to perform better. This leaves both the input and output of the tests and code unchanged, only the way the code processes input is altered. It is important in this step to edit nothing in the test code or the outputs or inputs. Otherwise, either the test would behave differently or new tests would have to be written. The latter goes directly against the TFD principle.

4.1.4 Test Driven Development

Test Driven Development is a combination of the previously mentioned techniques. A Unit Test is written before any code is, the test is then executed and should fail. After the failure, the most basic code to make the test pass is implemented. The test is then executed again and should pass. After this first pass, the code written for the passing of the test, and only this code, is edited to perform and look better. This follows all steps mentioned above, a *Unit Test* is written according to *Test First Development* and is *Refactored* later.

4.2 Continuous Integration

Continuous Integration (CI) is a software development technique in which developers upload the edits on the software, or their *build*, to a central server which then *continuously integrates* the code from multiple developers. This to prevent integration headaches when the code of multiple developers has diverged to such an extent that it would take much more time to make the edits work together than if they had been integrated early on.

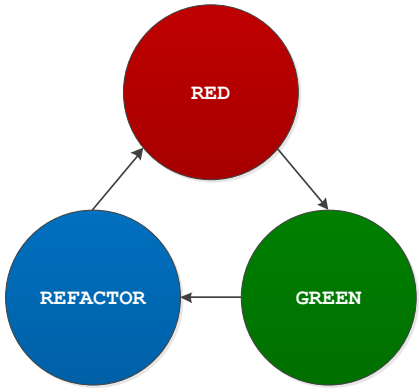

Figure 2: Three step TDD design flow

Combining all of these practices saves developers, and by extension the company they might work for, a lot of time and of money. Considering today's competitive market for software development, any edge that can be obtained is a plus. Even more so when plenty of free Continuous Integration solutions exist that employ open or widely used standards.

4.2.1 Revision Control

There are many aspects to a properly maintained CI system, but one key aspect of overall programming has to be Revision Control (RC). Not to be confused with the "undo" button in your preferred editor, a good RC system does allow for any and all mistakes made over different edits to be undone with very little work. There exist many systems for revision control, but they all have in common that they track changes one way or the other, and most importantly that these changes can be undone.

4.2.2 Build Automation

A useful but not required aspect is build automation. Using a timer or trigger, the build automatically runs with the latest updates, preferably from an RC system. This way, the binaries are always up to date and the developer does not need to wait for compilation to run the latest tests. Although the latter is less imported in a proper CI system as will be discussed further on.

4.2.3 Test Automation

As the code is built at scheduled times, and testing is needed regardless, it makes perfect sense to add a testing step to the automated build. Automating tests saves the developers yet another part of their time, thus freeing more for the actual development and debugging steps. A good CI system can read test reports, or at least some standard of reports.

4.3 xUnit

xUnit is a collection of frameworks that all follow the same basic principle. The xUnit specification defines several key components for a testing framework. They encompass an implementation of a *testcase* which contains a single unit test. These testcases are then enveloped in *testsuites*, which are groups of tests that need the same conditions before they can be executed. There are more

implementation details but these are not relevant to this thesis. JUnit is an implementation for the Java programming language of the xUnit specification. A useful part of the specification is a standard format for the test reports. The reports are written in eXtensible Markup Language (XML), which is a type of language used for formatting data. The JUnit format is supported by numerous (open source) tools, which makes it an ideal candidate for further investigation. [12], [13]

4.4 Python

Python is a high-level computer programming language that first appeared in 1991, with the third major revision being released in 2008. It supports object-oriented and structured programming and is easy to use as a scripting language. A great feat of Pythons community is that there are many (open source) libraries available for just about any function that comes to mind. This on top of the already impressive amount of libraries that Python itself supports. In addition to this, Python has all of its standard features explained in great detail with good examples in its online documenting system. The combination of these features allows any programmer, even with very little know-how, to quickly put together anything that comes to mind.

Part II
Developing a framework

Structurering!
Meer in detail gaan over methodes, voorbeeld van hoe framework gebruiken in plaats van details uit te leggen over de functies: design beslissingen
Meer grafische uitleg, workflow en dergelijke
In part I different aspects from both the software and VHDL development worlds were discussed in varying levels of detail. In this part bits and pieces of each subject are combined to form a new whole, a VHDL development framework.

5 Outlining

Before work begins on developing the framework, there need to be some boundaries set and goals to work towards. As VHDL development is done on both Windows and Linux, it only makes sense to keep the whole framework as platform independent as possible. To add to this, the framework needs to be developed in the short span of under a year by one developer who has had little experience doing so. The language chosen thus must be easy to master and work on all platforms with as few as possible modifications.

Furthermore, software practices will have to be applied to maintain clean code and a steady development process. To this end, work started with unit testing in mind. Parts of the VHDL testbench are separated to be executed independently in an automated fashion.

5.1 First draft

In section 3 some current industry practices were discussed and one of the very basic VHDL testing methods was found to be assertions. In order to remain as broad as possible, work started with processing these assertions first. The assertions thus need to be found inside the VHDL testbench, separated and saved in individual files to be executed.

One of the main problems that quickly arose was: *Where are the assertions located?*. To prevent overcomplicating things, the assumption was made that all tests are fitted inside their own procedure or function and that these would be called from inside the architecture body. All that would remain was to find the architecture body, find the keyword *begin* and extract every line from the body.

When the script was run on a file, a few problems quickly came in view:

- Tests can be a lot more complicated than first assumed
- Procedures and functions aren't that easily made to contain fully working code
- Writing tests this way is actually *more* time consuming

To try and counter the problems, using a standardized function library was proposed. This way, verified code could be integrated which reduced the workload when writing the testbench. Making use of this library also brought a degree of uniformity to the testbenches, which helped with readability.

5.2 First improvements

As mentioned above, one of the first improvements was a standardized function library. To be of any use, this library should contain functions and procedures that are either used extensively throughout the developmental process, have a great potential of being used or that, when created on their own, would impose a significant workload on the developer. Considering the large percentage of the industry still working in the VHDL-93 specification, the library would need to be compatible with this version. However, to ensure future usability and steer development towards the newer versions, i.e. 2002 and 2008, these should be supported as well.

A second improvement, yet undiscussed, would be maintaining a clean set-up. Throughout the first testing, it became apparent that when multiplying the amount of testbenches that are executed, the files around them are multiplied in the same way. Massive clutters of 'temporary' files and folders would quickly pollute the workspace, and thus keeping track and removing unneeded files would be of the essence, especially considering later modifications (see section 7.1).

5.2.1 The library

The first addition to the library was the tracking of signals and arrays (called *vectors* in VHDL). VHDL itself provides assertion-based verification, as mentioned before section 3.1. However, these assertions are long and nondescript. To make these easier to read and reproduce, functions could be made that act the same as a well-written assertion, but are much easier to read. Furthermore, assertions can be used to pass information about non-critical signals to a shell environment that looks for specific output text. An example:

```
procedure reportBack(
    b : boolean; result : string) is
begin
    if (not Q) then
        assert false report
            "test success" & ht & "name: "
            & "Checking Q" severity note;
    else
        assert false report
            "test failed" & ht & "name: "
            & "Checking Q" severity note;
    end if;
end procedure;
```

```
wait until rising_edge(clk);
reportback(('0' = Q), "Checking Q:1");
wait until rising_edge(clk);
reportback(('1' = Q), "Checking Q:2");
wait until rising_edge(clk);
reportback(('0' = Q), "Checking Q:3");
wait until rising_edge(clk);
reportback(('1' = Q), "Checking Q:4");
wait until rising_edge(clk);
reportback(('0' = Q), "Checking Q:5");
wait until rising_edge(clk);
reportback(('1' = Q), "Checking Q:6");
wait until rising_edge(clk);
```

In the left column, the source code of the function used in the right column is displayed. To maintain the same level of useful information (report on success *and* failure, name of test) without a procedure, the code would be excessively long. On top of that, the readability is improved greatly with only the logic and the message to display remaining. If this much improvement comes from a very simple function, it is not unlikely that a great deal more can come should the library be expanded.

5.2.2 Cleaning up

The second addition, simultaneous with the first, is the implementation of a clean-up system. As mentioned before, a massive amount of 'compiled clutter' is created when tests are separated into

multiple testbenches. Several ways of dealing with this were considered, such as:

- Tracking changes in the working directory
- Filtering needed files and deleting everything else
- Compiling in a specific folder and extracting what was necessary

Each of these methods has its advantages and disadvantages, but as a start the second method was chosen. This was straightforward to do, considering compiling occured in a local folder and all the used files were known. The disadvantage being that, if the files used should change and this is not implemented in the code, it would result in the loss of these files.

5.3 Code execution

With many files being generated and every file tested by hand, an automated compile and test system had to be devised. As discussed in section 3.5, ModelSim was deemed the most useful tool for this development. ModelSim supports command-line execution of its commands, which are even easier to invoke should the tool be added to the system's *path* variable. Python, being a script language, fully supports execution of command-line code with it's built-in function *os.system()*, an example:

```
1 os.system('vlib work')
2 os.system('vcom -2008 -work work tb_dff.vhd')
3 os.system('vsim -c work.tb_dff(Behavioural) -do "run -all;exit"')
```

In the above example, the VHDL library *work* is made, the file *tb_dff.vhd* is compiled to it and the architecture *Behavioural* of tb_dff is executed with the options *"run -all;exit"* (this means as much as *run from start to end, then exit the program*). Work is a standard name for a library, but it is prone to abuse, i.e. putting everything in the work library. Nevertheless, this is a script that is being executed and as such generic names aren't a problem for the script itself, only for debugging by the developer. The -2008 and -c flags indicate the VHDL-2008 version is being used and that the tool works in command-line mode.

To summarize, the script

1. Took a testbench
2. Separated its tests into different testbench files
3. Compiled and executed each of these files
4. Displayed the output in the console used to execute the script from

So some tasks are automated, but there still isn't any clear overview of results nor is there full automation.

5.4 Code organisation

The next step in the process was code organisation. Up until now everything has been done sequentially in the main Python file. With the code becoming increasingly complex and to increase the above promoted readability, functions were made to contain their own parts of the code. All of these functions initially assume full control of every variable in the file, a bad practice but needed. The function setup begins very traditionally:

1. Setup: Set all files and folders, generate unique name for current run
2. Parsing: Read source, sort into testbenches, ready commands for execution
3. Execution: Execute seperate testbenches and capture output
4. Clean-up: Remove everything except captured output

As is visible in the list, commands are now automatically captured in an output file with the following code:

```
readcmd = os.popen('vsim -c -quiet "work.' + entname
                    + '(' + line
                    + ')"   -do "run -all;exit"').read()
outputfile.write(readcmd)
```

The *os.popen()* command opens up a console window, whose contents are read in combination with the *read()* command that follows. The variables *entname* and *line* are the extracted entity name from the original source and a variable integer. This integer is the chosen name for the architectures from the different testbenches, who are simply named *1, 2 ...* . This proved to be somewhat illegible for different testbenches, especially for multiple runs, so it was quickly replaced with:

```
readcmd = os.popen('vsim -c -quiet "work.' + entname
                    + '(' + archname + str(test)
                    + ')" -do "run -all;exit"').read()
```

Where each architecture keeps its original name with simply a number appended to it.

5.5 Result evaluation

As is evident from the section 5.4, tests are executed and results captured automatically, but making sense of a whole lot of command line output from a text file is challenging. ModelSim makes this more difficult by adding a header and a lot of information that isn't really interesting at this time of development. Things such as which files were loaded prior to the execution and which preference file is being read are not needed. They might be in the event of severe failure but for a quick and easy overview of results, they are unwelcome.

A great deal of filtering of the output file was required, only the passed and failed testresults were needed for a proper report. In section 5.2.1, it is shown that asserts are passed as notes with the explicit words *test success* and *test failed*. A simple filtering on these words per line of the output file should return every testresult, neatly ordered in passed and failed groups. A total testcount can easily be tallied from the combination, and as such a crude report is written to a text file.

5.5.1 JUnit XML

Previously discussed in section 4.3, the JUnit implementation of xUnit uses XML for report viewing. The editor that was used throughout the development of the thesis is *Eclipse*, which is written in *Java*, the same language JUnit is an implementation for. So unsurprisingly, Eclipse readily supports JUnit XML reports. They can be read and displayed by manually navigating to them or editing the project properties to automatically read them.

To read an XML report, one would first would have to be generated. A very basic JUnit XML report consists of a *testsuite* and a number of *testcases*. A testsuite is a group of tests, and a testcase is a single test, in this case a passed or failed assert. The testsuite is named for the

unique testrun identifier. A string contains every testresult, both passed and failed, separated by a newline. Using a simple wordmatch and a regular expression search with the command *re.search()*, the information is extracted into a testcase class, per test. Finally, the resulting file is written to the correct file, composed of again the unique identifier and the clear word *testresults* in the current working directory.

As this proved to be too crude for a proper report, the decision was made to move to a pre-made, open-source Python implementation of a JUnit report, aptly named python-junit-xml.[13] This package requires its own external component to be installed, called setuptools [14]. In this thesis, versions 1.0 and 3.5.1 respectively were used, but versions 1.3 and 8.0.4 are currently available. The package supports more options than were first used, such as the testcase duration, the name of the suite parent and many more.

Again there is filtering on the words *success* and *failed*, but regular expressions are abandoned in favour of simple line splitting using the '-' token, as it was needlessly complicated. Test failures now feature some details on how or why they failed, with the *message* argument. Readability has improved as well, with each argument being defined before being used. Furthermore, generated reports are now able to be loaded into Eclipse, and by extension any other JUnit compatible viewer. This was previously not possible due to some missing options that were unable to be located.

6 Fresh start

As everything up until now was based on code created on the fly, it became clear that the original files weren't ready to be refactored or otherwise modified. The experience gained from writing everything mentioned in section 5 was put to good use, and a clean file was started. With this file, a number of assumptions were made:

- Everything should be organised into functions
- Functions should work independently
- Optional arguments to modify the behaviour should be specified
- Proper documentation should be provided
- A log detailing events should be held

All of these sound very logical to any experienced developer, and so there needed to be taken greater care of common software development practices to ensure the code would be understandable to outside developers. With this in mind, a division was made by functions that would represent the different steps in the parsing process.

6.1 Organising functions

The file was divided into parts that each held a certain functionality, disregarding specific details in favour of behavioural descriptions. From this, a number of functions were listed that should be able to get the entire job done, plus or minus some additions. These were:

- setup() Set up files, set all global vars, process cmdline arguments with argparse
- logwrite() Write to the log file: errors, completed jobs etc.

- get_path(path) Return absolute path if not already absolute path
- setup_parser() Prepare the parser to accept correct cmdline arguments
- make_tempdir() Create the temporary working directory
- parse_source() Grab source file, extract code, arrange functions and procedures
- test_format() Arrange found functions and procedures in their own executable files
- parse_tests Grab processed source/files, execute and capture output
- format() Grab output, format output
- xmlwrite() Grab processed output, convert to JUnit compatible XML file
- get_time() Extract the passed time from a ModelSim time notice
- cleanup() Remove temporary files and directories

Some of these bear striking resemblance to functions previously implemented. However, a lot of behaviour has been split into different functions, with as much useful standalones available. Some new behaviour has been brought up as well, such as the log and the tempdir functions.

6.2 Argparser

One of the features that was brought up was an *argument parser*. The idea of adding arguments to the command-line was long-existing, but only recently it was revealed that Python had a built-in argument parser. Using the library *argparse*, it would become possible to have a complex yet simple to use commandline interface with optional and required arguments. Included in the parser is also the option for a clear and thorough help, providing users with all the information they might need to run the script. An example:

```
1 p = argparse.ArgumentParser(description='VHDL testbench parser'
2                             , formatter_class
3                               = argparse.ArgumentDefaultsHelpFormatter )
4 p.add_argument('-c', '--cmd'
5                 , help = 'specifies script being called from commandline'
6                 , action = 'store_true', dest='cmdline', default=False )
7
8 arguments, unknown = p.parse_known_args()
```

First, the argumentparser *p* is defined, its options are a description and a formatter class. The formatter class is used to build the help when asked, it formats the output of all the *help* strings and the possible additions they need. Secondly, one or more arguments are added to the parser with the *add_argument* function. The options here are the flag(s), a description for the help, the action to be carried out (in this case store a boolean), the name of the action's destination and a default value. Only the flags are a required option, the rest are added for their usefulness. And finally, the arguments are parsed to *arguments* and *unknown*. The simple command is *parse_args*, but by using parse_known_args it is possible to also capture any unrecognized arguments without the parser forcefully exiting, these arguments are stored in unknown.

6.3 Log keeping

One important aspect of debugging is a properly maintained log. Before the refresh, log keeping was limited to reading the ModelSim command line information. No output from the script was displayed, unless haphazardly specified for crude debugging during development. With the creation of at least a basic log, it was possible to get a more detailed description and location of where things

might have gone wrong. Having the possibility of logging also immediately increases the desire to use it, as usage should be simple and results are immediate.

6.3.1 Basic log

To start off, a simple text file was created and its location might be decided through the use of a command line argument with argparse. Of course, to be readable, the log file should maintain a certain formatting. To be properly useful, a certain timekeeping aspect needed to be incorporated as well. The simple step by step plan would be:

1. Create log file if not already existent
2. Determine time of writing
3. Append message with time to the log
4. Close the file (proper file handling)

These steps combined give a basic but detailed and useful log. It should then give any debugging information that might be needed.

6.3.2 Buffering

It was quickly determined that the log was not usable until the script had already gone through several critical phases, each with very possible and critical messages that needed to be kept. A file couldn't be used before it was created, but it couldn't be created until the directory was set up which might require some logs being kept. There were two ways to deal with this, the first is turn off logging for all that comes before the log file is created; the second is to keep a buffer of log messages and to flush them to the log once it was ready. Seeing as it was easily implemented, the second option was chosen and a simple array was kept with log messages:

```
1 def logbuffer(level='n',message ='No message given.'):
2     global logs_buffer
3     if logstarted:
4         logwrite(level, message)
5     else:
6         logs_buffer.append([level, message
7                             , time.strftime("%Y.%m.%d - %H:%M:%S") ])
```

The variable *logs_buffer* is used to store the messages, the keyword *global* is required for Python to allow it to be modified. If the logs have started, indicated by the boolean *logstarted*, the original log write function is simply called. In this function, the time is kept for every logwrite and as such the buffer needs to keep this as well with the format Year.Month.Day - Hours:Minutes:Seconds, in the typical 'full' numbers YYYY.MM.DD - hh:mm:ss. Finally, a certain level is given as argument. This level indicates the severity of the log, inspired by the assertion severity levels in VHDL. Adding this provides the possibility of filtering a long log for critical information.

6.3.3 Actual log

With the buffer in place, the log could be properly expanded for some more advanced features. Priority is determining whether the log file had been created or not, which is done by using the global variable *logstarted*, mentioned in section 6.3.2. After creating the log, a header was written to the file containing information such as format of the messages used, the unique name of the test run and some basic cosmetics. All that remained to be done now was to write the buffered messages and ready the file for further additions.

6.4 Getting ready

Titel?

To ensure a proper work environment, temporary files and folders should be created, as well as more permanent files such as the log and results. At first, files were stored in a folder in the same directory as the source file. This was a perfectly acceptable practice to start off with, but eventually it became clear that users might take offence to folders seeming to appear in and out of existence while the script was running. Furthermore, to prevent the results and logs of being scattered to wherever the source files might be located, a central folder was put to use to store any and all files kept after the script had run its course.

6.4.1 Temporary directory

Considering that current-day operating systems have a a modern work environment, including folders for temporary storage and other out-of-sight stashing, it stood to reason to make use of this. Python has built-in functionality to access feature, in the form of *tempfile.gettempdir()*, which is a function that returns the temporary directory of the user calling the script. By creating a uniquely named folder within the temporary directory and doing all the work inside, it would be possible to simply remove this folder at the end of the run and leave a clean slate behind.

6.4.2 Results directory

With the platform independence in mind, there was a need for a folder that always stayed in the same location, regardless of the Operating System (OS). Most current-day OS's have a documents folder per user, and so good use could be made of this. With the Python function *os.environ['USERPROFILE']* referring to this folder, a folder aptly named *VHDL_TDD_Parser* is created that stores a new folder for each run of the script. However, if the script was called without the command line flag, one can assume it to be run by a more advanced program, perhaps a scheduler, that would automatically configure the files and folders to be stored in its workspace. Developers might also make use of this feature to disregard the central storage and have their results with the source files.

6.5 Processing source files

Originally, one file at a time was processed, with the testbench inside expected to contain several different test cases. As work progressed, it became obvious that a large VHDL project might contain several testbenches, and that each of these testbenches might contain multiple test suites with their own test cases. Work had to be done on maintaining a way of keeping different files separate, so that a proper trace could lead back to the original source, rather than a mix of everything.

6.5.1 Processing methods

Remembering the first draft in section 5.1, there were different ways of splitting the testbench into cases and suites. Considering the command line arguments were now available, it would stand to reason to give the developer the possibility of choosing in what way they desired to split their testbenches. Supporting every possible way is impossible, and as such the choice came down to three varieties:

- Start Stop identifiers
- Line per line
- Partitioned with identifiers

In the Start Stop method, every single line between certain keywords is assumed to contain a test. It is assumed that everything above the keywords is architecture header and everything below is a general 'end of the architecture body'. This method is the easiest to parse, look for the keyword and create a new testbench with every line. A disadvantage is that this is unsuited for large testbenches, with code duplication through the roof considering all processes need to be defined outside the keywords.

In the Line method, the parser looks for a process. This process should contain one test per line. It looks for this process itself and as such, can easier make a mistake than the Start Stop method.

Finally, in the Partitioned method, the developer indicates which blocks are testsuites by using the *--test* and *--end* keywords. Everything in between these keywords is assumed to be a single block of testcases, to be put in the same new testbench. As such, and unlike the previous two methods, it is possible to define multiple testsuites. This method gives the greatest advantages and, with some expansion, might enable a more intelligent handling of the testbenches that are being processed.

6.5.2 Formatting the testbenches

Now that the tests and other relevant information were extracted, the creation of the new testbenches was started. All parse methods from section 6.5.1 returned the same variables:

- File header
- Entity name
- Architecture name
- Architecture header
- Architecture footer
- List of found testsuites

Splitting the architecture header from the rest of the header allowed easier change of the architecture name, which was used in formatting the tests. For easier access, all of these variables were contained in a tuple which was then given to the formatting function. Using this tuple, a template was formed for the general testbenches. As the library functions mentioned in section 5.2.1 were used, a library header containing it was added to the header. In the next step, the architecture name was replaced with a unique name to allow each testsuite to be called generically as will be discussed in section 6.6. This generic name also including a *duplication avoider*, a counter to prevent multiple runs of the same source file creating conflicting names. The testsuites were encapsuled by the architecture header and footer, an everything was written to a file, one per test. Lastly, all of the unique names were then stored in a different tuple which was returned by the formatting function.

6.6 Execution

Titel?
With the

7 Surrounding programs

Titel?

7.1 Hudson-CI

The CI solution that was investigated is Hudson-CI which provides an extensive range of features, including everything listed above. The used features are:

- Timed and triggered building from an RC repository
- Automated testing of said build
- Humanly readable reports in the *JUnit* format
- Graphical and statistical overview of test progress throughout builds

8 The future of testing

Iets over VHDL features, verbeteringen etc

9 Conclusion

References

[1] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Springer US, 2000.

[2] Charles M. Weber, Neil C. Berglund and Patricia Gabella. "Mask Cost and Profitability in Photomask Manufacturing: An Empirical Analysis". In: *IEEE Transactions on Semiconductor Manufacturing* 19.4 (2006), pp. 465–474.

[3] Aldec Inc. *Active-HDL Configurations*. 2014. URL: https://www.aldec.com/en/products/fpga_simulation/active-hdl (visited on 19/12/2014).

[4] Cadence. *Incisive Enterprise Simulator*. 2014. URL: http://www.cadence.com/products/fv/enterprise_simulator/pages/default.aspx (visited on 19/12/2014).

[5] Xilinx. *Vivado Synthesis*. 2014. URL: http://www.xilinx.com/support/answers/62005.html (visited on 19/12/2014).

[6] Altera. *Quartus II support for VHDL 2008*. 2014. URL: http://quartushelp.altera.com/14.1/master.htm#mergedProjects/hdl/vhdl/vhdl_list_2008_vhdl_support.htm?GSA_pos=1&WT.oss_r=1&WT.oss=vhdl%202008 (visited on 19/12/2014).

[7] Altera. *ModelSim 10.0 release notes*. 2014. URL: http://www.altera.com/download/os-support/release-notes_10.0c.txt (visited on 19/12/2014).

[8] Martin Fowler. *UnitTest*. 2014. URL: http://martinfowler.com/bliki/UnitTest.html (visited on 30/06/2014).

[9] M. Siniaalto and P. Abrahamsson. "A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage". In: *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*. ESEM '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 275–284. ISBN: 0-7695-2886-4. DOI: 10.1109/ESEM.2007.2. URL: http://dx.doi.org/10.1109/ESEM.2007.2.

[10] L. Williams, J.C. Sanchez and E.M. Maximilien. "On the Sustained Use of a Test-Driven Development Practice at IBM". In: *AGILE 2007 Conference (AGILE 2007), 13-17 August 2007, Washington, DC, USA*. AGILE '07. IEEE Computer Society, 2007, pp. 6–14. ISBN: 0-7695-2872-4. DOI: 10.1109/AGILE.2007.43.

[11] B. Thirumalesh and N. Nachiappan. "Evaluating the efficacy of test-driven development: industrial case studies". In: *ISESE*. Ed. by Guilherme Horta Travassos, José Carlos Maldonado and Claes Wohlin. ACM, 2006, pp. 356–363. ISBN: 1-59593-218-6. URL: http://dblp.uni-trier.de/db/conf/isese/isese2006.html#BhatN06 (visited on 23/01/2007).

[12] J. Kivi et al. "Extreme programming: a university team design experience". In: *Canadian Conference on Electrical and Computer Engineering*. CCECE '00. Halifax, NS: IEEE, 2000, pp. 816–820. ISBN: 0-7803-5957-7. DOI: 10.1109/CCECE.2000.849579. URL: http://dx.doi.org/10.1109/CCECE.2000.849579.

[13] Brian Beyer. *Python JUnit XML*. 2014. URL: https://pypi.python.org/pypi/junit-xml/1.3 (visited on 07/05/2014).

[14] Jason R. Coombs and Phillip J. Eby. *setuptools*. 2014. URL: https://pypi.python.org/pypi/setuptools (visited on 07/05/2014).

Appendices

A Code examples

A.1 DFF testbench

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY tb_dff IS
END tb_dff;

ARCHITECTURE Behavioural OF tb_dff IS
    COMPONENT dff
    PORT(
         d   : IN  std_logic;
         clk : IN  std_logic;
         q   : OUT std_logic;
    END COMPONENT;

    SIGNAL d   : std_logic := '0';
    SIGNAL clk : std_logic := '0';
    SIGNAL q   : std_logic := '0';

    CONSTANT clk_period : time := 10 ns;
BEGIN
     uut: dff PORT MAP (
         d => d,
         clk => clk,
         q => q
         );
    clk_process : PROCESS
        BEGIN
             clk <= '0';
            WAIT FOR clk_period/2;
             clk <= '1';
            WAIT FOR clk_period - clk_period/2;
    END PROCESS;

    stim_proc: PROCESS
    BEGIN
        WAIT FOR clk_period;
        assert q = '0'
            report "Wrong output value at startup" severity FAILURE;
        d <= '1';
        WAIT FOR clk_period;
        assert q = '1'
            report "Wrong output value at first test" severity FAILURE;
        d <= '0';
        WAIT FOR clk_period;
        assert q = '0'
            report "Wrong output value at final test" severity FAILURE;
        WAIT;
   END PROCESS;
END Behavioural;
```