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

I Problem and background **1**

1 Problem **1**

2 Introduction **2**
2.1 Hardware Description Languages 2

3 Current industry practices **4**
3.1 Assertions 4
3.2 Coverage 4
3.3 Verification 5
3.4 Open Source VHDL Verification Methodology 5
3.5 Simulation tools 5

4 Software development & practices **6**
4.1 Test Driven Development 6
4.2 Continuous Integration 7
4.3 JUnit 8
4.4 Python 8

II Developing a framework **9**

5 Outlining **9**
5.1 First draft 9
5.2 First improvements 10
5.3 Code execution 11
5.4 Code organisation 11
5.5 Result evaluation 12

6 Fresh start **14**
6.1 Organising functions 14
6.2 Hudson-CI 15

7 The future of testing **16**

8 Conclusion **17**

List of Figures

1 Typical CMOS design flow 3
2 Three step TDD design flow 7

Part I
Problem and background

1 Problem

Developing VHSIC Hardware Description Language (VHDL), like any code, is prone to error creation, either by user or by wrong product specifications. To ensure errors are weeded out before the more expensive production begins, the code must be subjected to rigorous testing. Currently, large and impractical tests are needed to fully test a product.

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.

In this thesis, the objective is to provide VHDL users with an operating system independent framework. This framework will allow users to quickly and consistently create, modify and execute testbenches. To accomplish all of this, a number of tried and proven external tools will be gathered around a central Python script. This combination will ensure timely and automated building, testing and test report generation.

2 Introduction

2.1 Hardware Description Languages

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. 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) library provides a number of extensions on the original VHDL 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 D stands for Delay, and it simply puts on its output, Q, that which was on the input, D, one clock period earlier. The *architecture*, in this case Behavioural, takes the description of an entity and assigns a real implementation to it. 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.

This is a basic example of an entity, an architecture and a process. This 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. Therefore, it is paramount that all code on all levels is tested thoroughly, which is done by use of *testbenches*

[**bergeron00**].

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. If the device performs normally, the received output sequence should match a certain *golden reference*, the 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.

If a device is not tested properly and faults propagate throughout development, they can be very expensive to correct, especially at the stage of production, where a single photomask, used to "print" part of the layout, can easily cost $100,000 (€81,000 at time of writing)[**weber06**]. Therefore a large portion of time is spent writing and executing tests.

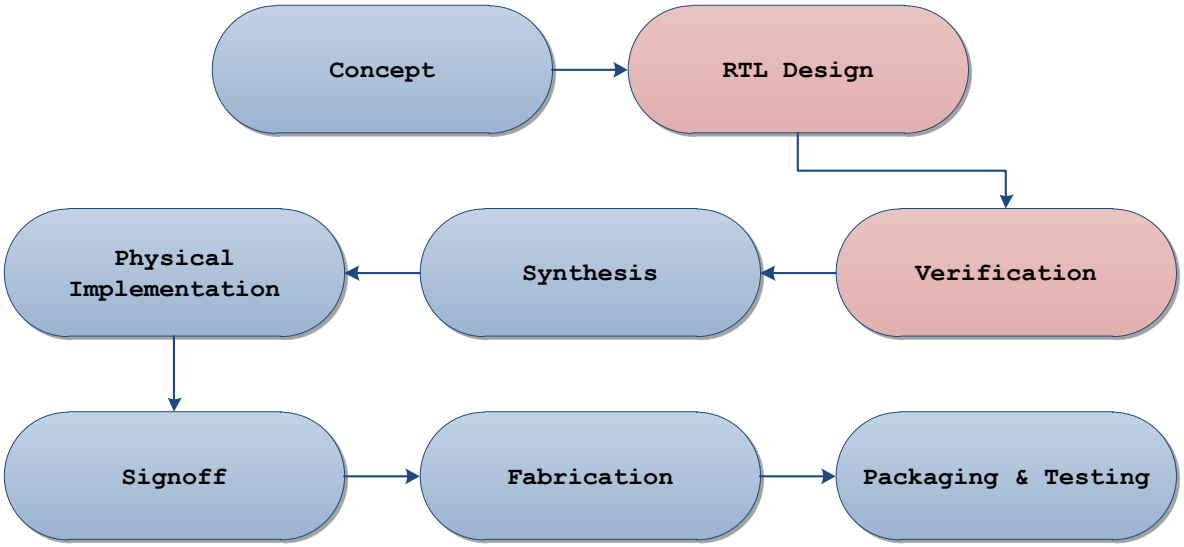

Figure 1: Typical CMOS design flow

3 Current industry practices

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 subsection 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 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.

3.2.1 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.

3.2.2 Functional Coverage

Functional Coverage (FC) is the practice of measuring whether design covers all possible use states it can possibly be in. Good practice is to include all possible cases, including corner cases, cases that cover multiple clock cycles, erroneous cases such as receiving an interrupt when processing or overflowing a buffer and so on.

3.3 Verification

3.3.1 Constrained Random Verification

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 wanted 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 and all behaviour, including the unexpected.

3.3.2 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 it out of use because proving large designs is not only tedious but takes up ridiculous amounts of man-hours.

3.4 Open Source VHDL Verification Methodology

The Open Source VHDL Verification Methodology (OSVVM) is a set of packages that makes it easier for CRV and FC to be implemented in a project. It consists out of two packages, *Random.pkg* and *Coverage.pkg*. The new feature, which it dubs Intelligent Coverage (IC), redefines its FC model based on holes in the FC coverage and randomization. The advantage is obviously that 100% coverage is always within reach, even when the original draft did not achieve full coverage.[**ICoverage**]

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, however, 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.

3.5.1 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. And last but not least, ModelSim is one of the industry's most used simulators, enabling a pool of experience to be consulted.

Figuurtje of vergelijking tussen simulators?

4 Software development & practices

During the making of this thesis, a number of approaches was investigated and some were put to practical use. In this chapter, the more useful and tried of the aforementioned are discussed in some 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[**VHDLUnit**]. This practice has proven to increase test coverage [**Siniaalto:2007:CCS:1302496.1302946**], decrease defect density [**TDDinpractice**] as well as improve code quality [**TDDinpractice**, **conf/isese/BhatN06**]. 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, an 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

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, 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.

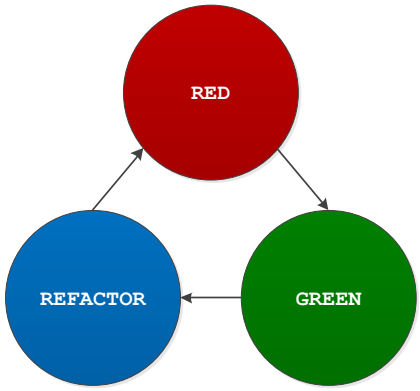

Figure 2: Three step TDD design flow

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.

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 build 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.

4.2.4 Hudson-CI

Combining all of these practices saves developers a lot of time and, by extension, the company they might work for, a lot 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 servers exist that employ open or widely used standards. The CI solution that was investigated is Hudson-CI. Hudson-CI 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.

4.3 JUnit

JUnit is an implementation for the Java programming language of the xUnit specification, which defines several key components for a testing framework. Considering the relevance here is not the framework itself but the format used for its reports, details about its implementation and use will not be discussed.
The JUnit 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, such as Eclipse and the aforementioned Hudson-CI, which makes it an ideal candidate for further investigation.

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, many (open source) libraries available for just about any function that comes to mind. This is on top of the already impressive amount of libraries 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

In part I different aspects from both the software and VHDL development worlds were discussed, each 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 bounds set and goals to work towards. As VHDL development is done in both Windows and Linux environments, it only makes sense to keep the whole framework as platform independent as possible. To add to this, the whole 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, making use of 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 would also bring a degree of uniformity to the testbenches, which would help 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 will need to 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 subsection 6.2).

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 subsection 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. It is easy to see that, 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 needed

Each of these methods has its advantages and disadvantages, but to begin the second method was chosen. This was easy to do, considering compiling happened in a local folder and all the files that were being used 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 test by hand, an automated compile and test system had to be devised. As discussed in subsection 3.5, ModelSim was deemed the most useful tool for this development. As many tools do, ModelSim supports command-line execution of its commands, even easier to invoke should the tool be added to the system's *path* variable. Python, being a script language, fully supported 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 now took a testbench, separated its tests into different testbench files, compiled and executed each of these files and 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 was being done sequentially in the main Python file. With the come become 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 previous subsection, tests are executed and results captured automatically, but making sense of a whole lot of command line output from a text file isn't easy. ModelSim makes this 'worse' by adding a header and a whole lot of information that isn't really interesting at this time of development. Things such as what files were loaded prior to the execution and what 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 needed, only the passed and failed testresults were needed for a proper report. In subsection 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

As mentioned in subsection 4.3, the JUnit implementation of xUnit uses XML for report viewing. The editor 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, we would of course need to have one generated. As mentioned above, testresults are stored in a text file, which is readily accessible. 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. A crude implementation of how to parse the testresults is show below:

```
xmlpath = os.getcwd() + os.sep + target + '_testresults.xml'
xmlfile = open(xmlpath, 'w+')
xmlfile.write('<testsuite tests="' + str(totaltests) + '" name ="'
```

```
                + target + '_tests">')
for line in everyline.split('\n'):
  words = line.split(' ')
  if line.find('success') != -1:
    xmlfile.write('\n\t<testcase classname="' + target
                  + '" status="' + str(words[0]) + '" name="'
                  + re.search('(?<=name: )(.*)(?= -)', line).group(0)
                  + '"/>')
  elif line.find('failed') != -1:
    xmlfile.write('\n\t<testcase classname="' + target + '" status="'
                  + str(words[0]) + '" name="'
                  + re.search('(?<=name: )(.*)(?= -)', line).group(0)
                  + '">')
    xmlfile.write('\n\t\t<failure> ' + line.split(' - ')[2]
                  + ' </failure>\n\t</testcase>')
xmlfile.write('\n</testsuite>')
```

The testsuite is named for the unique testrun identifier *target*. The string *everyline* 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 needed information is extracted into a testcase class, per test. Finally, the resulting file is written to the *xmlpath*, 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.[**junitxml**] This package requires its own external component to be installed, called setuptools.[**setuptools**] In this thesis, versions 1.0 and 3.5.1 respectively were used, but versions 1.3 and 8.0.4 are available at time of writing. The package supports more options than were first used, such as *time_taken*, which is the testcase duration, the name of the suite parent and many more. The implementation is shown below:

```
xmlpath = os.getcwd() + os.sep + target + '_testresults.xml'
xmlfile = open(xmlpath, 'w+')
test_cases = []
for line in everyline.split('\n'):
    words = line.split(' ')
    if line.find('success') != -1:
        time_taken = get_time(line.split('-')[-1][7:])
        name = (words[0] + ' - '
                        + line.split('-')[1].split('name:')[1].strip())
        print name
        test_cases.append(TestCase(name, target, time_taken, None))
    elif line.find('failed') != -1:
        time_taken = get_time(line.split('-')[-1][7:])
        name = (words[0] + ' - '
                        + line.split('-')[1].split('name:')[1].strip())
        message = ("-".join(line.split(' - ')[2:-1])).strip()
        tc = TestCase(name, target, time_taken, None)
        tc.add_failure_info(None, message)
        test_cases.append(tc)
ts = TestSuite("Test Suite", test_cases)
xmlfile.write(TestSuite.to_xml_string([ts]))
```

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 started with limited knowledge and experience of Python, it became clear that the original files weren't easy 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 was to be organised into functions
- Functions were to work independently
- Optional arguments to modify the behaviour were to be specified
- Proper documentation was to be provided
- A log detailing events was to be held

All of these sound very logical to any experienced developer, however it was the inexperience that led to this point of rewrite. As such, 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

First it was chosen what the file was to be able to do, not how it was going to be done or what the exact implementation would be. From this, a number of functions were listed that should be able to get the entire job done, plus or minus some additions. They were:

- setup() Set up files, set all global vars, process cmdline arguments with argparse
- logwrite() Write things to the logfile: errors, completed jobs etc.
- get_path(path) Return absolute path if not already absolute path
- setup_parser() Prepare the parser to accept correct commandline arguments
- tempdir() Create the temporary working directory
- parse_source() Grab sourcefile, extract needed code, arrange functions & procedures
- test_format() Arrange found functions & procedures in their own executable files
- parse_tests Grab processed source/files, execute & capture output
- format() Grab output, format output
- xmlwrite() Grab processed output, convert to JUnit compatible XML file
- cleanup() Remove temporary files & 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, the tempdir

6.2 Hudson-CI

Sectietje over hudson (nodig om te verwijzen)

7 The future of testing

Iets over VHDL features, verbeteringen etc

8 Conclusion