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

UNIVERSITEIT
GENT

Preface

Abstract

Contents

List of Figures

List of Tables

Part I
Problem and background

1 Problem

Developing 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 is subjected to rigorous testing. For full product testing by conventional means, large and impractical tests are needed.

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 engineers and software engineers to find and correct errors with minimal delay and maximal effort. This process should affect the least amount of code possible as to minimize time spent retesting and recoding.

In this thesis, a number of mechanics are used to optimize both testing and coding. Based loosely off of Test Driven Development (TDD), tests are written to both function and be tested independently to maximize test coverage with minimal effort. To this end, a library with often used functions and other useful code is made available. Alongside of it is a tool, written in Python, to process tests made with this library independently and represent the results in a quick and easy to read format.

2 Introduction

2.1 Digital Electronics

There are two kinds of electronic appliances and circuits, digital and analogue. Digital electronics differ from analogue electronics in that they use a discrete set of voltage levels to transmit signals. The most common number of items in the set is 2, a level for one (commonly named "high") and a level for zero (commonly named "ground" or "low"). The advantage of using a discrete number of levels rather than a continuous signal as is used in analogue electronics is that noise generated by the environment, thermal noise and other interfering factors, will have but a minor influence on the signal.

To process these discrete signals, electronics are made up of transistors that nowadays are formed in with the *Complementary Metal Oxide Semiconductor* (CMOS) technology. This technology uses both an *NPN* and a *PNP* transistor that work in a push-pull configuration. The p's and n's in NPN and PNP simply stand for *Positive* and *Negative*, they are made of positive and negative doped lumps of semiconductor (usually Silicon-Dioxide or SO_2). A transistor is basically a blockade on a track and depending on the force applied to its Gate, it opens or closes the track.

In reality, the force takes form of a current and an NPN transistor opens its gate when a positive current is applied. A PNP transistor, however, always leaves its gate open until a current is applied. This means that if we send the same signal to an NPN and a PNP transistor, with one of the signals inverted, we can open and close two parts of the entire circuit at the same time. This is useful to both direct a certain signal to ground and at the same time close its connection with the *source* (the power source). Hence also the name *Complementary* MOS, the NPN and PNP complement each other.

A certain combination of transistors is used to make *logic gates.* These logic gates make it so that only a certain combination of ones and zeroes at the inputs result in certain ones or zeroes at the outputs. For instance, one of the most common logic gates is a *nand* gate (a *not and* gate). This gate has a number of inputs ranging from 2 to theoretically infinity (but practically 3 or 4) and only outputs a low signal if all of the inputs are *high* (digital one), otherwise its output is *at ground* or *low* (digital zero). The other most common logic gate is the *nor* gate (a *not or* gate). This gate outputs a low signal if any of its inputs are high, otherwise it outputs a low.

A common mistake is to think that low or ground mean *zero voltage.* This is only partially true, the high signals are measured with ground as their reference. So a high signal of 1.8 Volts would be 1.8 Volts higher than ground, and could be considered to be at 1.8 Volts if ground is the theoretical zero.

A certain combination of these logic gates are used to build higher-level blocks such as flip-flops, which are used to make registers and so on up to the entire chip design.

2.2 Hardware Description Languages

A *Hardware Description Language* (HDL) can be used to describe any one of these levels, right down to the logic gate level, however this last one might not be a good idea considering most synthesis tools can produce superior logic gate-level layouts[1]. The level that uses certain blocks of logic gates to describe more complex behaviour is called the *Register Transfer Level* or RTL. Some blocks are standard implementations that have been widely used and nearly fully optimized, such as memories, flip-flops and clocks. An RTL flip-flop implementation 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 IEEE library provides a number of extensions on the original VHDL code 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 or DFF. The D stands for Delay, and it simple puts on its output Q that which was on the input one clock cycle 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 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. It waits for the rising edge of the clock, which is a transition from zero to one, and then it schedules the value of D to be put on Q until 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 named *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 somewhat difficult to pinpoint the exact level and location of any errors. Therefore it is paramount that all code on all levels is tested thoroughly, this is done by use of *testbenches.*[2]

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 performs a certain sequence of 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 also interesting to see how well a device performs if its inputs behave outside 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.

It is easy to see that if a device is not tested properly and faults propagate it 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[3]. Therefore a large portion of time is spent writing and executing tests[4]. Practices to improve both the speed and quality of testing and coding exist in a large number, but the focus chosen in this thesis is *Test Driven Development* (TDD). This practice has proven to increase test coverage[5], decrease defect density[6] as well as improve code quality[7, 6].

3 Mission and objectives

The goal of this thesis is to ease development and testing of VHDL code, with a focus on testing and test reporting. Practically this means the development of several tools which can be independently used and each have their own merit.

- A testbench parser that ensures independent testing and proper test report generation.
- A library with many widely-used functions as to speed up programming, as well as functions to make reporting possible.
- A test report that can easily be read but still holds a significant amount of detail on errors and successes.

References

[1] http://www.asic-world.com/vhdl/intro1.html

[2] Writing Testbenches: Functional Verification of HDL Models

[3] Mask Cost and Profitability in Photomask Manufacturing: An Empirical Analysis

[4] Citation needed !!

[5] A comparative case study on the impact of test-driven development on program design and test coverage

[6] A Longitudinal Study of the Use of a Test-Driven Development Practice in Industry

[7] Evaluating the Efficacy of Test-Driven Development

Part II

Appendix