Metropolia

Todor Vlaev

Agile Testing on an Embedded Field Programmable Gate Array Platform

Helsinki Metropolia University of Applied Sciences
Bachelor of Engineering
Information Technology
Thesis
6 May 2011

Metropolia

Abstract

Author(s) Title Number of Pages Date	Todor Vlaev Agile Testing on an Embedded FPGA Platform 30 pages + 3 appendices 6 May 2011
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Embedded Engineering
Instructor	Anssi Ikonen, Principal Lecturer

Agile software methodologies are the state of art methodologies used on current software projects. Testing is one of the main pillars of agile development and many of the practices are common among various flavours of the methodologies. Despite their wide-spread adoption in different domains, agile testing practices still seem to be a novel concept on embedded programming projects. This is specifically true when it comes to hardware design modeling. Thus, the goal of this project was to introduce the main concepts of agile testing and demonstrate their application on an Field Programmable Gate Array (FPGA) platform.

The project was conceptually divided into two parts. The first one was the design and implementation of an FPGA development board. The second part focused on developing hardware design modules with a suitable hardware description language and ultimately building a contained testing system to demonstrate the most important agile testing practices.

The result of the first phase was a working FPGA development board and an Ethernet extension board. During the second phase example hardware models were designed with MyHDL. Unit tests were implemented before the actual modules, thus adopting a test-driven development (TDD) approach. The tests were automated with the help of a continuous integration server. A viable process for a functional testing routine was also outlined.

Based on the outcomes, it can be concluded that agile testing practices can be successfully utilized even in the specific domain of digital design. The natural continuation of this project would be the implementation of the suggested functional testing routine.

Keywords	agile testing, test-driven development, FPGA, hardware modeling, MyHDL, continuous integration, test automation

Acknowledgments

I would like to express my deepest gratitude to Anssi Ikonen and Dr. Antti Piironen who were my mentors during the years I spent at Helsinki Metropolia University of Applied Sciences.

Contents

1 Introduction

Requirements often change significantly over the lifetime of a technology project. It may be a result of change in the client's business processes, new technological breakthroughs, organizational restructuring and other reasons. [1] Situations such as these are especially relevant to the field of software development.

Traditional software development practices established in the past assume a set of well-defined phases of the whole project such as design, implementation and testing. These practices, generally known as 'waterfall', rely on the assumption that requirements are fixed and the process of producing software is predictable. Unfortunately, most of the time, both of the assumptions are wrong. This is the main issue that the so-called agile software methodologies aim to address. [2] Extensive testing regime is an integral part of these methodologies and therefore they have introduced different practices which are commonly referred to as 'agile testing'.

Software development for embedded systems has its peculiarities compared to traditional methods. Typically, an embedded system is considered to be a resource-constrained computer system with a specific function, for example a mobile phone or a washing machine controller.

Digital design by itself is considered a separate field of embedded development. Field Programmable Gate Array (FPGA) chips implement digital circuit designs modeled with so-called hardware description languages (HDLs) [3,21]. In other words, HDLs are effectively used to create hardware modules. Therefore, rarely is digital design with HDL considered software development, even though in my opinion it clearly is.

Having introduced the main concepts, the goal of the project is to demonstrate the use of agile software testing practices in the very specific field of digital design with HDL. The project is inspired by my personal experience with agile development and the realization that many of the practices are general in nature when it comes to technology and particularly programming projects.

2 Agile Software Methodologies

Agile testing, as the term suggests, relates to agile software methodologies. Even though, in my opinion, many of the testing practices presented in the current paper do not necessarily require an agile process in place, it is important to explain the principles of agile software development in order to gain insight into the full benefits that the practices are able to provide. This chapter presents the key notions behind agile software methodologies (also simply referred to as ‘agile development’ or ‘agile methodologies’) and an overview of a popular agile methodology, namely Extreme Programming (XP).

2.1 The Agile Manifesto

In 2001 a number of established names in the agile world forged the principles of agile software methodologies and embodied them in the so-called agile manifesto which is presented in the following quotation [4,27].

> Individuals and interactions over processes and tools
> Working software over comprehensive documentation
> Customer collaboration over contract negotiation
> Responding to change over following a plan
>
> That is, while there is value on the items on the right, we value the items on the left more [5]

The agile manifesto is well known and often cited in related literature. The principles quoted above lay the foundations of the agile software development philosophy. However, these are not rules carved in stone and methodologies may place different weight on the four aspects of the manifesto.

2.2 Extreme Programming Values

There are five values that XP emphasizes. [1] They are easy to comprehend and strike as obvious. However, as this fact may be misleading to people who adopt the software methodology for the first time, the XP community has ensured that each of the values is well explained and justified. Figure 1 depicts the five XP values. The following paragraphs provide further insight into their meaning and the reasoning behind them.

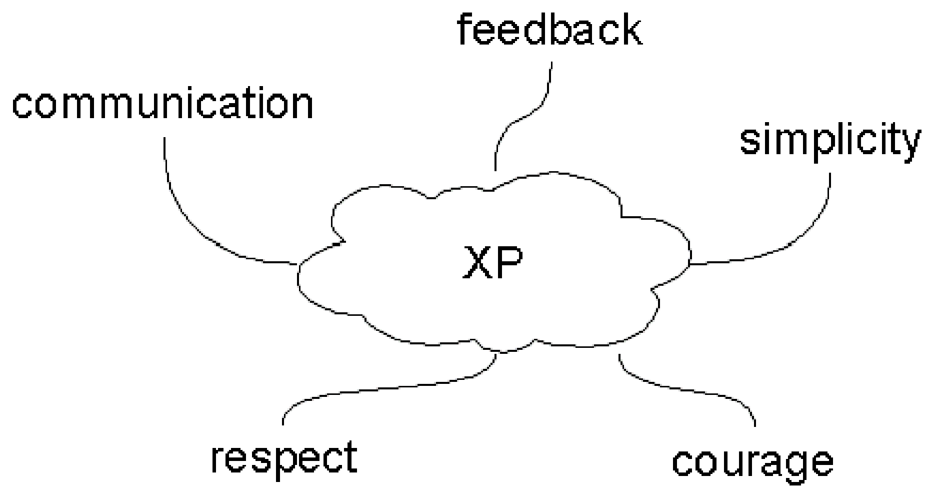


Figure 1. Extreme programming values. Adapted from Holcombe (2008) [1,21]

Research has shown that the reason for most failures of software development projects is breakdowns in communication. XP acknowledges this issue and identifies three levels of communication that require particular attention. The first one is the communication among the clients themselves. It must be ensured that they respond adequately to their business needs in setting the requirements for the project. The second tier is the client-developer channel because it is vital that programmers understand the business value certain functionality brings. Finally, the communication among the development team is crucial for the success of the project. At all times all of the members must be involved in planning and decision-making. The rest of the XP values aim at supporting this third communication level. [1]

Feedback represents a specific aspect of communication that is worth outlining. At all times developers should seek active feedback from the customers. [1] This provides for a natural self-alignment of the project goals in a way that brings the best value to the customer. Additionally, developers should be constantly aware of the quality of their progress and the general view of their commitment to the common goals [1].

In the field of technology, especially software development, it is common that novel solutions emerge with fast pace. This fact is naturally tempting for the software engineer who is always eager to learn and experiment. However, in most cases using cutting edge technology could be not only adventurous but also threatening to the success of the project. Therefore, the XP value of simplicity matches well the famous Einstein quote: "Everything should be made as simple as possible but no simpler". [1]

Due to the fact that XP is a methodology that differs significantly from the traditional methods of software development, it requires that people have the courage to initiate and sustain change. Naturally, for good team relationships, respect is of indispensable value. [1]

2.3 Extreme Programming Activities

There are 12 basic practices of XP [1]. To a large extent they are typical of most of the agile software development methodologies. The following paragraphs present the activities which XP is most famous for. The practices related to testing are discussed in greater detail in chapter 3.

Pair-programming is probably the most well-known XP practice. It means that two developers work on a single workstation (one keyboard and one screen) simultaneously. At the same time one of the programmers is coding, the other one is inspecting the code. Collaboration is highly valued. As the pair discusses issues, more ideas are generated and the probability of introducing faults in the program diminishes. Pairs change on a regular basis and an additional implication is that developers gain an overall picture of the whole application as they work on different parts of the source code. If properly managed, team relations and communication can be heavily improved. [1]

The on-site customer XP practice recommends if possible to locate a customer in the developers' premises. This will naturally improve the client-developer communication. However, it may also introduce certain complications if the client operates in a fast-changing business and he or she eventually becomes disconnected from the company's operations. [1]

The planning game and system metaphor are interrelated practices. At first the developers collect the so-called stories from the client during the planning game. A story describes a single piece of functionality that the application should implement. Based on a set of base stories, a skeleton of application architecture is built, which is referred to as the system metaphor. [1]

Collective code ownership refers to the fact that the source tree belongs to all of the developers. Any programmer is allowed to work on different parts of the code. This approach also assumes that there is a more constructive spirit once a fault is found, due to the fact that the responsibility is shared. [1]

The rest of the core 12 XP practices are presented in the following list:

- test-first programming
- small frequent releases
- simplest solution approach
- continuous integration
- coding standards
- refactoring
- 40-hour week [1].

It must be pointed out that all of the 12 practices bear the same importance and are an integral part of the XP process.

3 Agile Testing

The following is an overview of the most common practices used in agile software projects. It is important to note, however, that the techniques have not necessarily emerged together with the advent of agile development methodologies. Many of the ideas existed and the practices were applied well before it. However, it has not been until the surge of popularity of agile development that they received such widespread adoption as today.

3.1 Software Testing Levels

Before agile testing practices are introduced, it is essential that the different levels of software testing are clarified. Figure 2 depicts an interpretation of the popular V-model of software development.

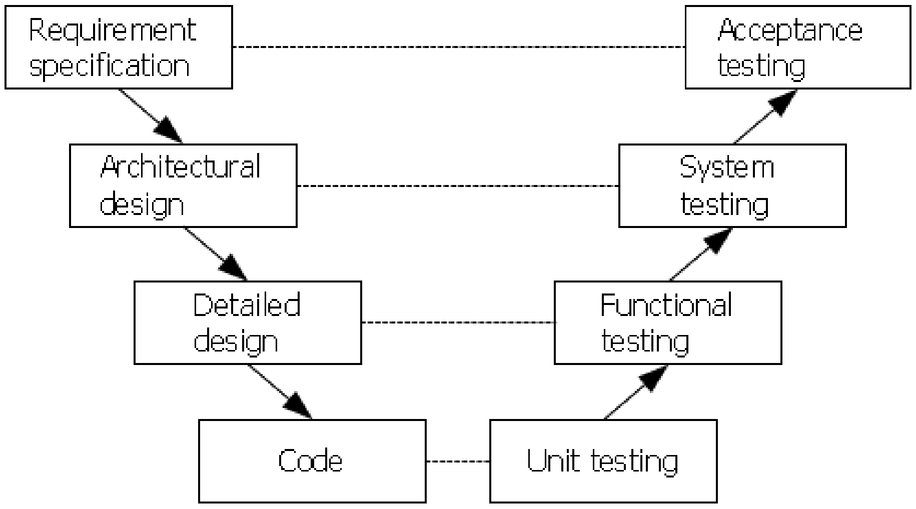


Figure 2. V-model of software development. Modified from Fewster and Graham (1999) [6,7]

The V-model is often referenced in relation to the waterfall model of software development [1,7]. Nevertheless, in my opinion, the V-model is an excellent representation of the different abstraction levels of software testing in general. Depending on the context, there may be more stages. This paper focuses mainly on the lowest two levels, namely unit testing and functional testing.

At the top of the V-model, acceptance tests verify the software against the customer requirements. Successful acceptance testing must assure that the system under test fulfills its purpose and brings the desired customer value. Tests at this level should be meaningful to the end users of the system and often mimic direct end user actions. A relevant example for a web application would be the ability of the user to log in.

System testing verifies the correct operation of the architectural entities of a system, for example, the communication between different servers once a user has issued a login request from a website. In the context of telecommunications, system testing could test the interworking between the different network nodes.

Functional tests examine the operation of a single design unit, for example a software component, or a single piece of functionality that spans over several components. Compared to system testing, functional testing is more focused on a particular feature and its integration within the developed product, rather than the whole system the product operates in. An example could be verifying database transactions triggered by user activity upon login.

At the lowest level of software testing, unit tests are implemented. This type of tests verifies the programming logic of the smallest software modules such as functions or class methods. For example, unit tests may test the software module validating the syntax of a username upon registration.

3.2 Test-driven Development

Test-driven development (TDD) is the practice of writing tests before writing the actual source code. Initially, this approach sounds counter-intuitive as the traditional workflow of software development has been to first implement and then test. [7,32] The TDD step cycle is presented in figure 3.

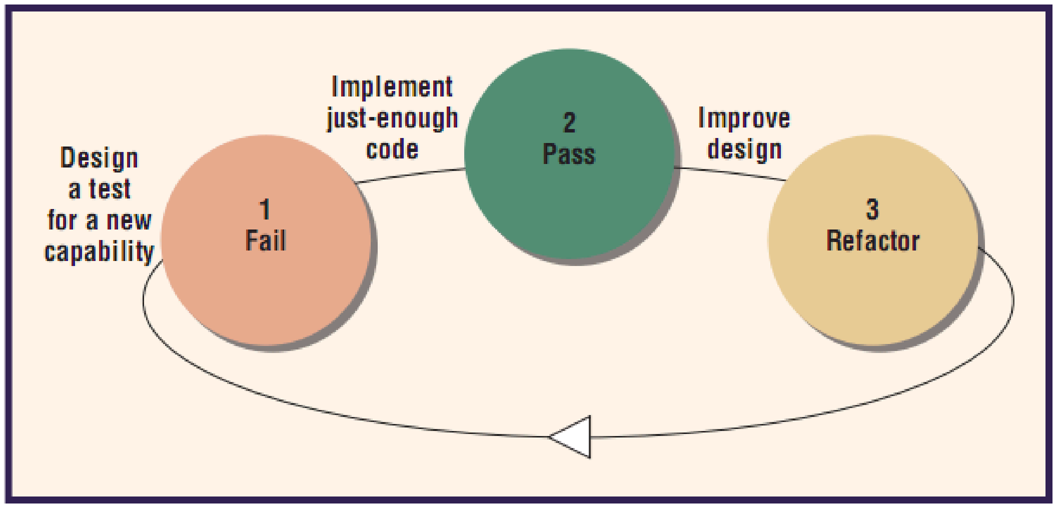


Figure 3. TDD step cycle [8,25]

The first step in the TDD cycle is to implement a test which inevitably fails as the implementation is not in place yet. The software developer then implements the required functionality in order for the test to pass. Ideally, as illustrated in figure 3, so much code is written as to satisfy the requirements of the test – no more, no less. Finally, the source code may be improved if needed and the functionality verified with the existing test.

It must also be pointed out that TDD may be applied at any of the described software testing levels as long as there is a clear set of requirements that must be fulfilled. The implemented tests will fail until the system under test satisfies the requirements. Additionally, as tests are implemented before the functionally, the pass-fail rate may provide useful progress metrics.

There are several significant benefits which justify the test-driven development approach. Already the first step of designing a failing test prompts programmers and testers to focus on the actual use of the source code to be written [8,26]. Along with the rule to implement 'just-enough' code, this enforces the goal of development and improves customer-orientation.

With time, writing tests for each piece of functionality as the software product develops produces a comprehensive test set. This reduces the risk of refactoring activities which may break functionality. Additional implication is the improved sense of security in the software developers of future changes in the code base. [7,33] Creativity and innovation are boosted as programmers need not fear so much to experiment and can easily verify the results of their work.

3.3 Test Automation

The idea of test automation is to delegate as much as possible of the manual testing performed by software testers to computers. That is not to say that computers can replace people. The goal is to offload the most laborious, repetitive and clerical work from testers to machines. Intellectual tasks such as identifying tests scenarios and designing the actual test scripts are still performed by software testers. At this point in time the tasks performed by the so-called test automation frameworks in general include execution, comparison and reporting. [6,17-18]

The greatest benefit of a test automation regime is running more tests more often with less time [6,9]. This is especially valued in the context of regression testing. Regression testing is an activity which verifies that new versions of the software have not introduced defects in pre-existing functionality. Running automated regression tests does not require much effort, as the tests have already been implemented [6,9].

Often, tests cannot be performed manually, for example, performance tests which would require millions of actions per second from the tester. On the other hand, some tests that take a prohibitively long time and are not meant to be executed often do not justify the effort to be automated. Such considerations have to be regularly made when automating tests. [6]

A typical automated test would have to perform several tasks:

1. pre-processing
2. test actions and dynamic comparison
3. post-execution comparison
4. post-processing [6].

Often the system under test must be set up before the test can be executed [6,176]. For example, a web application test would need to start and configure a browser, or create entries in a test database.

Once the system is set up, test scripts must reproduce the test scenario envisioned by the test engineer. For, example a test may want to verify that a user is able to log in by using the correct username and password. The test scripts should then fill in the correct fields and press a 'Login' button. If a 'Welcome' page should be opened, the test script could verify this by checking the page title – this is referred to as 'dynamic comparison', that is a comparison which is made during the test execution [6,107].

After the actual test has been completed, the system should be brought to its pristine state from before the test started. The actions required are known as post-processing [6,176]. Just before the post-processing the test scripts may want to examine internal log files for errors. Since this comparison is done after the actual test run, it is referred to as 'post-execution comparison' [6, 108]. Finally, the test status (pass or fail) and relevant artifacts such as logs or created files. must be collected and reported [6].

Test automation is a challenging activity which, however, is an indispensible part of current software development. Among the greatest challenges are maintaining testware (such as reference data, scripts and reports), building scripts which can handle unexpected errors, and creating quality tests.

3.4 Continuous Integration (CI)

Continuous integration is an activity that aims to integrate the software components of the product after each software change. This avoids the painful integration cycle after months of development. [9,xx] Continuous integration produces a build which ideally comprises several activities such as compiling the source code, inspecting it against quality criteria, deploying the application and running tests on the target system [9,4]. The results are then reported to the developer. Figure 4 presents the core parts of a CI system.

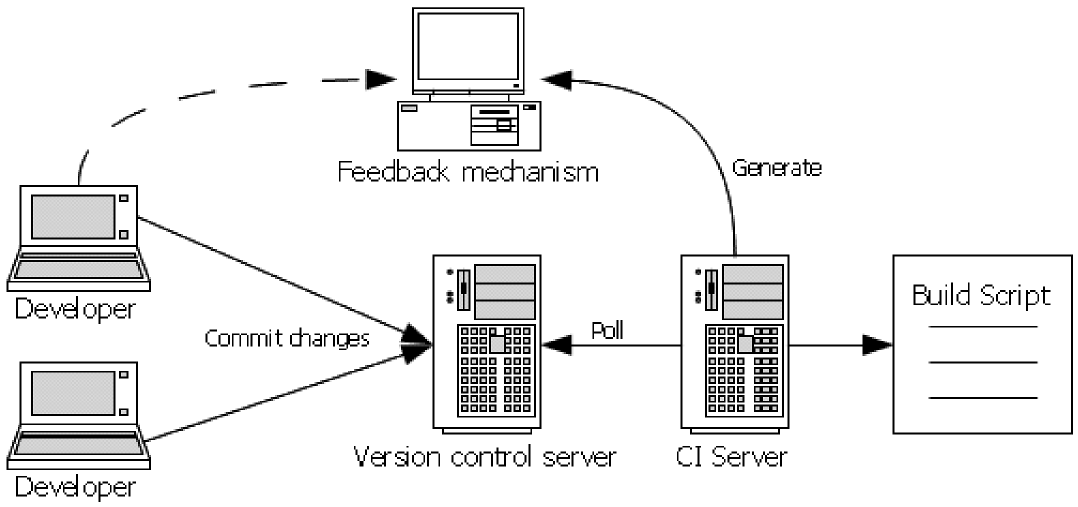


Figure 4. The core componets of a CI system. Modified from Duvall (2007) [9,5]. OpenOffice Draw computer shapes by Lautman (2010) [10]

A version control repository which manages the changes made to all software assets is a prerequisite for CI [9,7]. The action of introducing changes to the artifacts managed by the repository uses the term ‘commit’.

The CI server is polling the version control server for recent changes to the code base. Once a change is detected, the CI server executes the build script which performs most of the actions involved in integrating and testing the build. After the completion of the build script, activities results are stored on the CI server and reported in a suitable format, for example a web page. [9,5]

From a software developer perspective the flow has several steps. First, the software must be integrated locally, so that the developer has confidence that his or her changes will not break the common build. Then the programmer commits the changes and waits for the results report. Therefore, it is also crucial that the integration tasks performed by the build script do not take too much time. Time-consuming activities, such as running a full regression test suite could be executed during a nightly build. In case the results from the CI server indicate errors, the developer must correct the issues and apply the changes as soon as possible, so that the work of other people is not blocked. [9,5-10]

4 Digital System Design Methodology

Digital design refers to the design of digital circuits which need to fulfill specific functional requirements while at the same time adhere to constraints such as cost, performance or power consumption. The design of complex systems is made possible by a layer of logical abstractions, the most significant of which is representing information in discrete form. [3, 1-3]

4.1 Programmable Logic Devices

The traditional approach to implementing digital designs on integrated circuits has been the so-called application-specific integrated circuits (ASICs). ASICs provide the best speed, die size and power consumption metrics. To put it simply, the reason is that ASICs implement only specific functionality which, once embedded on the integrated circuit, cannot change. However, the same fact presents some significant drawbacks such as increased design time and inability to correct possible design defects or update the functionality. [10,249]

In contrast to ASICs, programmable logic devices (PLDs) are integrated circuits which include reconfigurable general-purpose logic resources. This allows for rapid design cycle and the ability to update the digital design when the product is already in production. The price for this flexibility is reduced speed and increased power consumption as compared to ASICs. Nevertheless, PLDs have become popular solutions because they are still able to deliver high performance with a significantly shortened time to market. [10, 250]

Complex programmable logic devices (CPLDs) and field programmable gate arrays (FPGAs) form the mainstream of PLDs on the market. CPLDs are a viable solution for simple control applications. These ICs offer lower logic density (number of logic elements) and speeds compared to FPGAs. Demanding applications which require heavy data processing are addressed by FPGAs. Both types of devices provide many different features within their segment and may be programmed one or multiple times. [10,255-257]

4.2 Design Flow

Due to the wide range of specific applications design projects must implement, it is difficult to establish a standardized design methodology [3,439]. Nevertheless, figure 5 attempts to define the most important phases of the digital design flow.

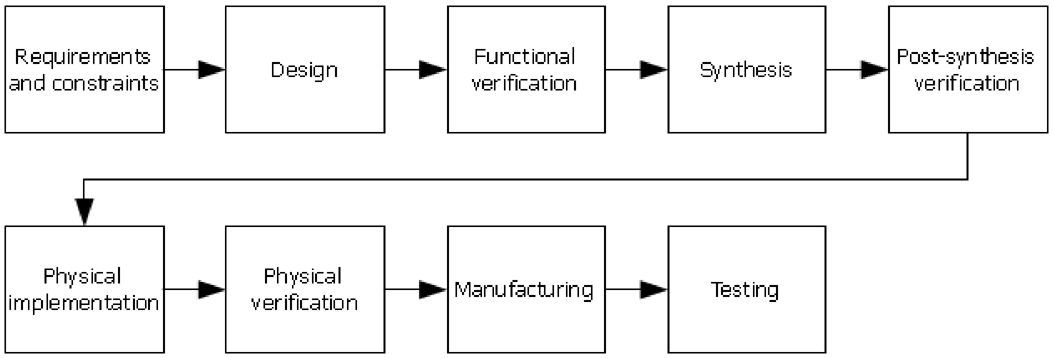


Figure 5. Digital system design flow. Modified from Ashenden (2008) [3,28]

Failures which occur in any of the post-design phases and before manufacturing cause a transition to the previous phase or, if a design defect is identified, to the design phase; this has been omitted from figure 5 for the sake of simplicity. The actual digital circuit modeling occurs during the design phase. The digital logic is then verified usually using logic simulators. [3,28]

Synthesis is the process of converting and optimizing the high level model of a digital system to produce a detailed structural model. This lower level of abstraction reached after the operation is the one of logical gates such as AND or OR. The structural model, also referred to as 'refined design', is essentially a description of the interconnection between primitive logic elements. Synthesis is performed by computer aided design (CAD) tools. [3,29-30]

The physical implementation and verification phases depend on the implementation fabric, such as ASIC and FPGA. There are, however, several common steps which must be performed. First, the circuit resources such as logical element type and count needed to implement the design must be determined. This is known as ‘mapping’. Then the exact place and route of interconnecting lines are specified during a so-called placement and routing phase. After mapping and placement and routing have taken place, more detailed estimates of system properties, such as propagation delays and power consumption, are made which aid the final physical verification. [3,30-31]

The output files of the physical implementation phase are used in manufacturing for actual implementation of the design. On an FPGA chip this usually means programming the device. It is then that actual tests (and not simulations) can be executed against the running system. [3,31]

4.3 MyHDL

Hardware description languages (HDLs) are used to model digital designs. HDLs are much like standard programming languages but tailored for the specific field of digital design. The most prominent HDLs are VHDL and Verilog with other alternatives such as SystemC and C++. The main differences between HDLs are in their more advanced features whereas there is a high level of similarity of the basic features. [3,21]

MyHDL is an open source Python package which allows Python to be used as a hardware description language [11,3]. Python is an open-source interpreted high-level object-oriented programming language. It is often labeled as a ‘scripting language’ because of its ease of use and rich set of utilities. Python is geared towards developer productivity and software quality [12,5-8].

There are several significant reasons why MyHDL was chosen for this final year project, even though the package is still under development. MyHDL utilizes many of the benefits of Python. Thus, it promotes rapid prototyping and the use of modern software development practices, such as unit testing, in hardware modeling. The key feature is that with certain limitations the MyHDL code can be converted into VHDL or Verilog, which allows to implement digital designs on PLDs. [11,3]

MyHDL is based on two Python features: generators and decorators. Generators are objects which can essentially be used as resumable functions. Thus, by using the generator's method `next()` it is possible to return values based on the previous state of the object, as it is retained. A decorator is a piece of syntax placed in front of a function declaration and is used to convert a function into a callable object. Decorators are used in MyHDL to create specific generator objects depending on the type of hardware model, such as sequential or combinational logic. [11,6] Listing 1 presents a simple example of an incrementer module with asynchronous reset signal.

```
ACTIVE_LOW, INACTIVE_HIGH = 0, 1

def Inc(count, enable, clock, reset, n):
    """ Incrementer with enable.
    count -- output
    enable -- control input, increment when 1
    clock -- clock input
    reset -- asynchronous reset input
     n -- counter max value
    """

    @always(clock.posedge, reset.negedge)
    def incLogic():
        if reset == ACTIVE_LOW:
             count.next = 0
        else:
            if enable:
                count.next = (count + 1) % n

return incLogic
```

Listing 1. MyHDL incrementer with asynchronous reset design [11,26]

In listing 1 `@always` is a decorator used with sequential logic, and the one used with combinational logic is `@always_comb` [11,86-87]. The decorator is creating the generator object and specifies the so-called sensitivity list of the module. The signals on the sensitivity list cause the generator to resume operation, in other words, to produce the next output value [11,57]. The next value of the signal is determined by setting its `next` attribute.

In order to convert the design to VHDL, it is necessary to create an instance of the incrementer module and the signals used by it. [69-70] Listing 2 shows just how simple this is. Once the Python script executes `toVHDL()`successfully, the VHDL files are created.

```
m = 8
n = 2 ** m

count = Signal(intbv(0)[m:])
enable = Signal(bool(0))
clock, reset = [Signal(bool()) for i in range(2)]

inc_inst = Inc(count, enable, clock, reset, n=n)
inc_inst = toVHDL(Inc, count, enable, clock, reset, n=n)
```

Listing 2. Converting the incrementer MyHDL design to VHDL [11,67-68]

The most important class is visible in listing 2 and is namely `intbv`. This class implements an integer-like type which introduces operations such as indexing and slicing that Python does not provide. Additionally, `intbv` is designed so as to relieve designers from the burden that integer representation issues pose in VHDL and Verilog. The MyHDL-provided type handles integers very similarly to standard programming languages. [11,13-18]

The MyHDL manual provides comprehensive information as to what exactly comprises the so-called convertible subset. The limitations apply to the code of the generators. Better flexibility is available for testbenches and pure modeling scenarios where the full power of Python may be unleashed. [11,57] In general the allowed types are integer and boolean with certain variations of lists. Even though this may sound quite restrictive, in my personal experience the language feels more flexible than VHDL.

5 Design and Implementation of the FPGA Board

An embedded FPGA board was designed for the purposes of the project. The work name of the board is Poart (the leading 'p' is intentional). The ideology behind Poart is to create an FPGA board with a minimum amount of circuitry for the chip to run and provide extension ports so that it can be used for various applications by designing an extension board. Additionally, an Ethernet extension board was implemented to facilitate the development of some type of network application.

5.1 Poart

Simplicity is one of the core values of Poart. This can already be seen from the block diagram presented in figure 6.

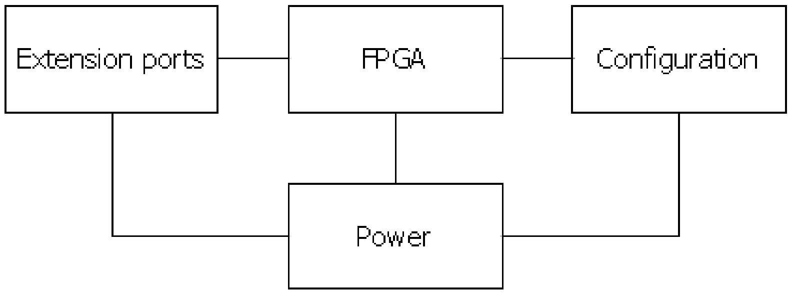


Figure 6. Poart block diagram.

The core of the development board is the EP2C8Q208C8N FPGA manufactured by Altera and part of the company's Cyclone II family of devices. The device was chosen because of my familiarity with Altera products. The Cyclone II FPGAs target low-cost embedded and digital signal processing (DSP) applications [13,17]. The highlights of the chip used on the board are the following [13]:

- 8256 logic elements
- up to 138 user input/output pins
- 165 888 total RAM bits
- 18 embedded multipliers

- 2 PLLs (phase-locked loops)
- 8 clock inputs
- maximum clock frequencies of 320 MHz.

The Cyclone FPGA can generally be configured (programmed) by various means. For the purpose of this project, first an on-board serial configuration device was programmed which then automatically configured the FPGA. The serial configuration device used non-volatile memory which allowed for the design to be retained even if the board is powered down, as the FPGA relies entirely on SRAM cells, in other words, volatile memory.

The power block in figure 6 provides the necessary power voltages to all of the other blocks of Poart. Finally, all of the pins which do not have a special purpose related to power or configuration are connected to the extension ports.

Appendix 1 provides a brief hardware guide to Poart, including more detailed information about, for example, power source requirements, clock inputs and extension port connections.

5.2 Ethernet Extension Board

The Ethernet extension board connects to one of Poart's extension ports. The board provides an Ethernet controller and a LAN port with the required magnetics. The core of the extension board is the Microchip ENC624J600 Ethernet controller. Its most important features are presented below:

- Integrated MAC and 10/100Base-T PHY
- Auto-negotiation supported
- SPI or parallel control interfaces
- 24Kbyte transmit/receive SRAM buffer
- CRC generation and security engine blocks
- +3.3V power supply voltage [14,1].

The extension board is designed so as to allow for the utilization of all of the control interface modes. The interface mode can be set by several SMD resistors placed on the top side of the printed circuit board. Further details about the board are presented in appendix 2.

5.3 Printed Circuit Boards

Mentor Graphics PADS CAD software was used to design the schematics and produce the PCB layout. The PADS autorouter was extensively used during the routing phase. There are settings which specify how rigorous the phases of autorouting are. The autorouter proved indispensible in routing LVDS (low voltage differential signal) pairs of traces which pose requirements such as equal trace lengths.

Despite the sophisticated capabilities of the PADS autorouter, manual routing was still required. The problem arises from the fact that not all of the significant information for the types of traces could be provided to the autorouter. Prioritization of routing order did help. However, clock and power signals still required manual optimizations. Nevertheless, the performance of the autorouter was impressive and in my personal opinion shortened the PCB design time by several folds. The printed circuit boards are shown in figure 7 below.

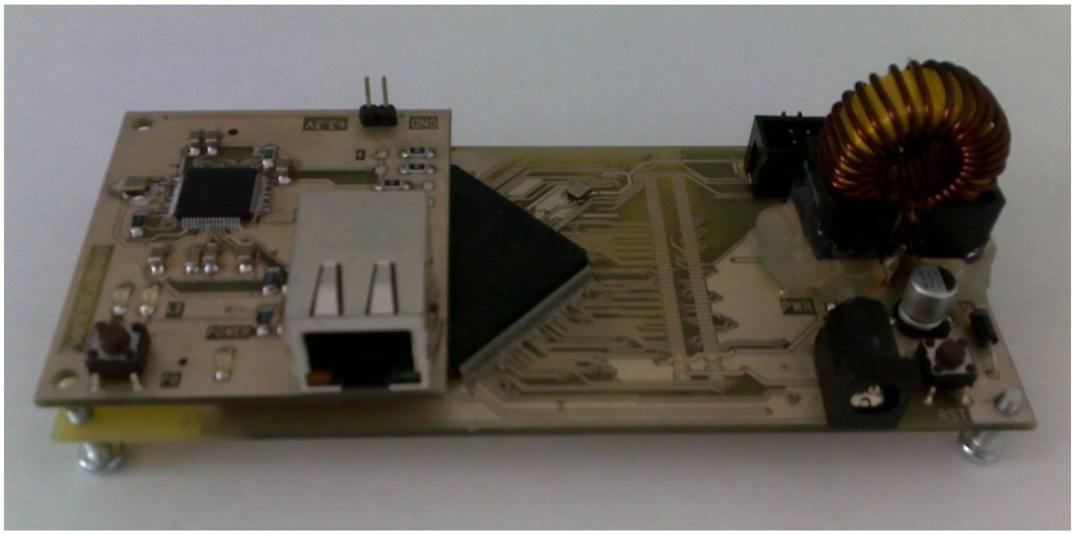


Figure 7. The Ethernet extension board connected to Poart

6 Development and Testing on the FPGA Board

6.1 Quartus

Altera provides a full development environment for their programmable logic devices. The free version of the software is called Quartus Web Edition. Compared to the licensed product it supports fewer device families and lacks some advanced features such as incremental compilation. [15] Nevertheless, the shortcomings of Quartus Web Edition (referred to simply as 'Quartus') are marginal in the context of small projects.

Quartus by itself provides all of the tools required for a successful design flow – from synthesis to programming an actual device. The basic steps are to create a project in the Quartus environment, specify details such as target device and top-level entity name, compile the design and finally configure the PLD. After the full compilation process various analysis reports are generated which may provide crucial information for the live design. [16,464]

The synthesis tool supports VHDL, Verilog, as well as Altera-specific formats, such as the so-called Block Design Format which is generated by a Quartus schematic drawing utility [15,463]. Namely the support for VHDL and Verilog provides the entry point for MyHDL to the design flow. Converted MyHDL designs are ready to be immediately included as Quartus project resources.

The Quartus environment provides a graphical integrated development environment (IDE). All of the functionality is available also through a command-line interface. [16,739] This allows for flexible configurations with scripts and ultimately makes the case of continuous integration.

6.2 Unit Testing

Probably the immediate and most obvious advantage of MyHDL is the ability to utilize the unit testing frameworks available for Python. The module 'unittest' even comes bundled with Python and is the standard unit testing framework used with it and inherently MyHDL. Nevertheless, there are significantly more alternatives available and for this project the so-called ‘py.test’ framework was used. The reason was that MyHDL’s creator Jan Decaluwe suggests in one of his MyHDL examples, that py.test could be promoted as the unit test framework of choice for MyHDL in the future. [17]

Poart’s extension board provides a pushbutton, but without a debouncing circuit. It is possible to design a hardware module with MyHDL that debounces the signal. Such a module may have different parameters such as sampling frequency, active level or enable signal. Let us assume that the enable signal is asynchronous and should force the output of the module to its inactive state. As this is a fairly small piece of the functionality a unit test can be written (in the spirit of TDD before the actual implementation is in place). A code snippet in appendix 3 presents the unit test. It also brings to light another advantage of MyHDL – designs may be written in such a way that they can be instantiated several times with different parameters, essentially producing different modules.

Due to lack of time the interface to the Ethernet controller of the extension board was not implemented as originally planned. However, the unit testing capabilities provided by MyHDL would have been an indispensable tool when modeling the design. Additionally, a trace of the signals can be generated with MyHDL for debugging purposes.

6.3 Functional Testing

Despite the fact that there was no networking application implemented on Poart, as the development and testing tools were available this scenario was still considered. When it comes to functional testing, it would be necessary that a test controller interfaces with the board, for example by sending a packet and verifying if the received response is correct.

In order to design and run automated tests, a test automation framework is needed. In my professional experience I have got acquainted with Robot Framework and found that it was a viable choice also for this project. Robot Framework is keyword-driven and general purpose in nature [18]. It is open-source and becoming increasingly popular. Figure 8 presents the high level architecture of the framework.

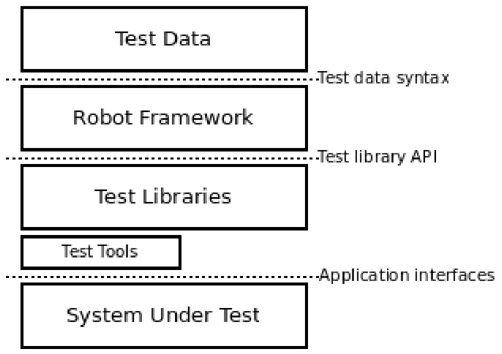


Figure 8. Robot framework architecture. Reprinted from the Robot Framework User Guide (2011) [18]

Robot Framework specifies its own test data syntax for implementing keywords. The framework does not interface directly to the system under test (in this case this is Poart) but does it through test libraries. The test libraries themselves could be wrappers for the tools that do the actual work. [18] For example, in order to test a protocol that Poart implements, it would be necessary to use a test tool such as a network traffic generator (it should be noted that here I refer to a software application, not a separate piece of hardware).

6.4 Test System Architecture

Figure 4 in section 3.4 already presented the main concepts of a CI system. Naturally, the setup required to implement CI and test automation for Poart differs in several details. The most obvious implication is that tests must be run against the embedded board. Thus, an entity that connects to the board, configures it and runs the tests must be present. This entity can be called a 'test controller'. The CI server instructs the test controller as to what test cases to run and then collects the results and any additional artifacts. The test automation framework and any specific tools, for example network traffic generators, should be installed on the test controller. Since configuration of the device must also be done through the controller, it is required that the Quartus environment is also installed. Figure 9 illustrates the architecture of the test system.

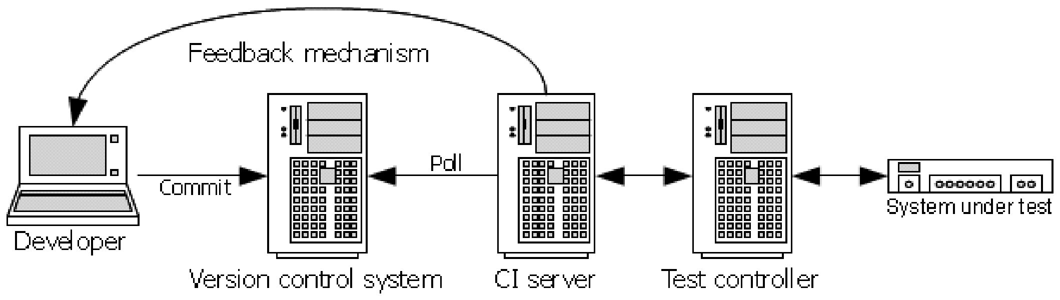
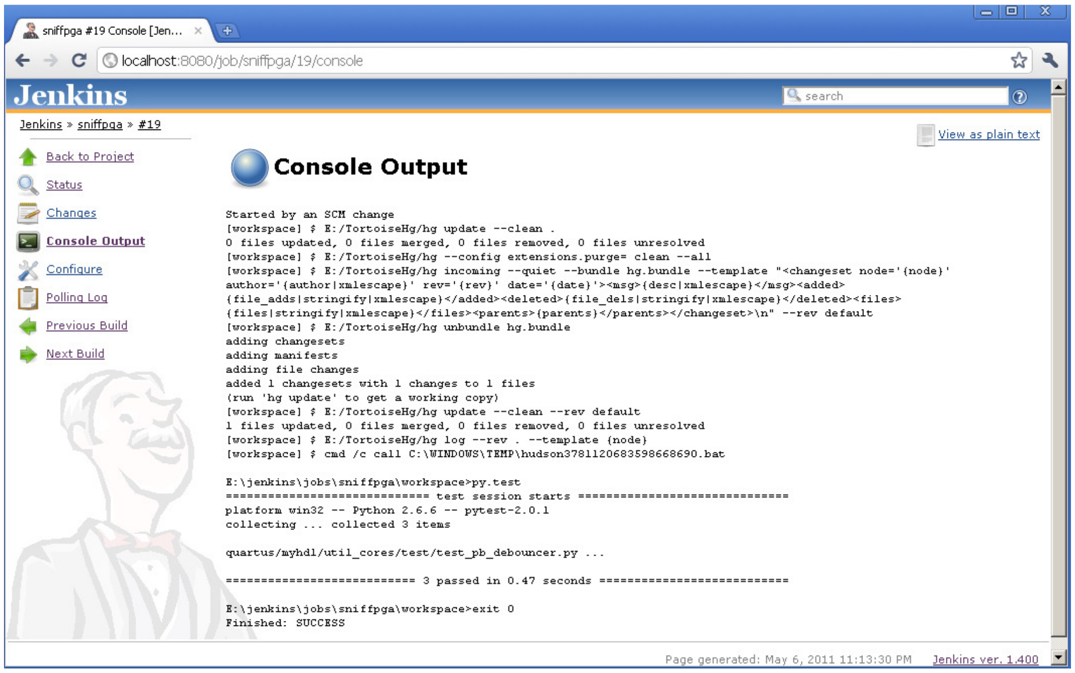


Figure 9. Test system architecture. OpenOffice Draw computer shapes by Lautman (2010) [10]

It must be pointed out that the developer's workstation and servers presented in figure 9 refer to logical entities which may well be present on the same physical machine. In fact for the purpose of this project the development workstation and CI server resided on a single virtual machine. There is nothing that prevents the same machine to host the version control repository and act as a test controller as well. In fact this was the original plan for the system. Nevertheless, in a real industry project most often separate machines perform the different functions.

The project was hosted on the Google Code website because it provided free storage and access to a Mercurial version control repository [19]. The CI server used on the project is Jenkins which is a prominent open-source project [20]. Jenkins, as already mentioned, was installed on a local virtual machine where all of the development took place. Thus it was not necessary to set up a web server. If the server is installed on a remote machine, it must be assured that the Quartus environment is also installed, so that Jenkins is able to compile the project. The installation process was extremely easy and Jenkins must be acknowledged for it.

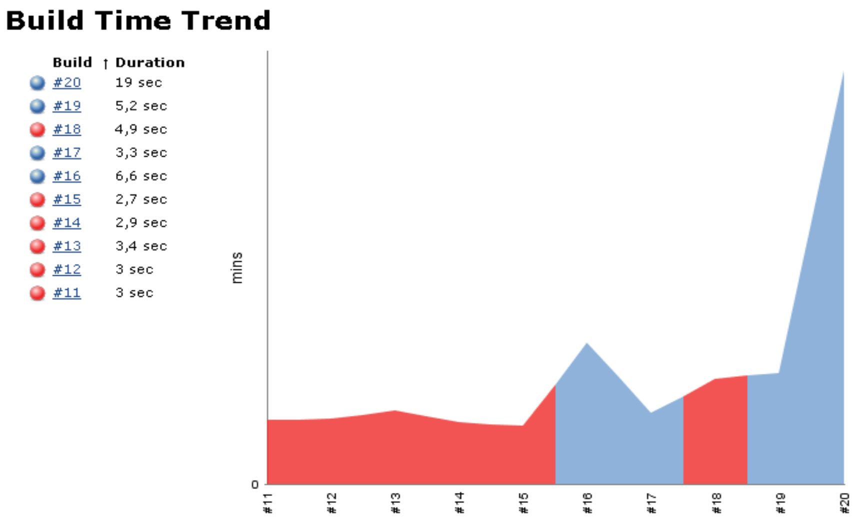
Jenkins is accessible and configurable through a web interface. The CI server was configured to poll the Mercurial repository every minute. An alternative would be to configure a periodical build. However, the 1 minute polling cycle provides the fastest feedback. If a change is detected, Jenkins pulls the newest version of the repository and triggers a build using the build script configured for the project. As an example, a pushbutton module and several unit tests were written in MyHDL. The build script simply calls py.test to execute the unit tests. Figure 10 presents a screenshot of the console output after a successful build.



```
Started by an SCM change
[workspace] $ E:/TortoiseHg/hg update --clean .
0 files updated, 0 files merged, 0 files removed, 0 files unresolved
[workspace] $ E:/TortoiseHg/hg --config extensions.purge= clean --all
[workspace] $ E:/TortoiseHg/hg incoming --quiet --bundle hg.bundle --template "<changeset node='{node}'
author='{author|xmlescape}' rev='{rev}' date='{date}'><msg>{desc|xmlescape}</msg><added>
{file_adds|stringify|xmlescape}</added><deleted>{file_dels|stringify|xmlescape}</deleted><files>
{files|stringify|xmlescape}</files><parents>{parents}</parents></changeset>\n" --rev default
[workspace] $ E:/TortoiseHg/hg unbundle hg.bundle
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
(run 'hg update' to get a working copy)
[workspace] $ E:/TortoiseHg/hg update --clean --rev default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
[workspace] $ E:/TortoiseHg/hg log --rev . --template {node}
[workspace] $ cmd /c call C:\WINDOWS\TEMP\hudson3781120683598668690.bat

E:\jenkins\jobs\sniffpga\workspace>py.test
============================= test session starts ==============================
platform win32 -- Python 2.6.6 -- pytest-2.0.1
collecting ... collected 3 items

quartus/myhdl/util_cores/test/test_pb_debouncer.py ...

=========================== 3 passed in 0.47 seconds ===========================

E:\jenkins\jobs\sniffpga\workspace>exit 0
Finished: SUCCESS
```

Figure 10. Console output of a successful build in Jenkins

If the changes break the unit tests, py.test would return an error visible from the console output and the build would be marked as failed. Jenkins, as well as any CI server, provides information about build history such as duration and status. This is illustrated in figure 11. In relation to this the number of builds that are kept is also configurable. This is important as in bigger projects storing the build artifacts may require a significant amount of disk space.



	Build	↑ Duration
🔵	#20	19 sec
🔵	#19	5,2 sec
🔴	#18	4,9 sec
🔵	#17	3,3 sec
🔵	#16	6,6 sec
🔴	#15	2,7 sec
🔴	#14	2,9 sec
🔴	#13	3,4 sec
🔴	#12	3 sec
🔴	#11	3 sec

Figure 11. Build duration and status trend in Jenkins

In the graph presented in figure 11, red signifies failed builds and blue successful builds. This is extremely useful information which provides insight into the 'health' of the code base.

A full setup to test Poart would require the following steps:

- execute unit tests
- convert MyHDL code to VHDL
- compile the Quartus project
- program Poart
- execute functional tests.

The process of executing the unit tests with the latest version of the code has already been explained earlier in this section. However, it is also important to highlight that the unit tests do not depend on anything else, for example it is not required to compile the design before executing the tests, nor do they produce any input used by the other phases. This means that unit tests may be executed in parallel to the other activities and thus save time. The simplest way to accomplish this is to configure a separate independent build in Jenkins.

The converting of MyHDL code and the compilation of Quartus must happen sequentially only after the first one had completed. In a small project where the conversion is a relatively short process, it would be practical that these two activities are executed in one build. It is then possible to configure that upon successful completion, another build is triggered which programs the board. Similarly, after successful programming, a final build can be triggered which executes the functional tests. By separating these activities in different builds it is easier to identify at which phase an error occured.

7 Discussion

Testing is one of the main pillars of agile development. However, unfortunately it is not uncommon that the first encounter with testing practices for fresh graduates occurs in the field rather than in the university classrooms. Therefore, the information on agile testing presented in the current paper provides a good start for further exploration to students who are about to commence a career as a software developer, be it on embedded platforms.

In fact the practices introduced are so general in nature that, in my opinion, they can be utilized in any type of programming projects. The field chosen for the project, namely digital design, was particularly challenging as it is not generally perceived as a software development activity due to the fact that the final product is a hardware design, rather than a program. It was therefore rewarding to demonstrate that by using open source tools, such as MyHDL, Jenkins, and Robot Framework, agile testing practices can be easily implemented in a digital design project. It is then also inherently true that embedded programming projects which use general purpose programming languages, such as C, could integrate even better.

As the project proceeded, greater emphasis was put on agile testing and practices. Nevertheless, a significant part of the work involved the implementation of the underlying hardware platform. The experience was fulfilling and the outcome is a functioning extensible FPGA platform which may be used in a variety of applications and which challenge the capabilities of automated testing even further.

The main challenge over the course of the project was the one of focus. The initial goals set for the reference application (the one to be tested) were too high and this occasionally led to frustration and procrastination. Nevertheless, in my opinion the delay and the subsequent outscoping of the originally aspired hardware design did not prevent me from achieving the goal of the project.

8 Conclusions

The goal of the project was to present the basics of agile testing and ways that the related practices can be applied in the field of digital design. For this purpose a small test system was successfully built to illustrate the key components of continuous integration and test automation.

Tools that facilitate agile testing are constantly evolving. The hardware description language MyHDL is an innovative project which allows for a more seamless integration with these tools than VHDL or Verilog. Based on Python, MyHDL promotes the use of modern software development practices in hardware modeling, such as test-driven development and unit testing.

The outcome of the project was a successful demonstration that agile testing can be integrated with the design flow of digital systems. Unit tests were automated and executed by a continuous integration server. The flow of the system was the same as what would be expected of any other more common application.

Future development of the system would require the implementation of functional tests by utilizing a test automation framework. The challenges lie in the integration with other testing tools such as network traffic generators and digital acquisition devices. The required steps were outlined in the report.

References

1 Holcombe M. Running an agile software development project. Hoboken, New Jersey: John Wiley & Sons.; 2008.

2 Pfleeger SL, Atley JM. Software engineering: theory and practice. Upper Saddle River, New Jersey: Pearson Education; 2006.

3 Ashenden PJ. Digital design: an embedded systems approach using VHDL. Burlington, MA: Morgan Kaufmann; 2008.

4 Larman C. Agile and iterative development: a manager's guide. Boston, MA: Pearson Education; 2004.

5 Beck Kent, Beedle Mike, van Bennekum Arie, Cockburn Alistair, Cunningham Ward, Fowler Martin, Grenning James, Highsmith Jim, Hunt Andrew, Jeffries Ron, Kern Jon, Marick Brian, Martin Robert, Mellor Steve, Schwaber Ken, Sutherland Jeff, Thomas Dave. Manifesto for Agile Software Development [online]. Agile Manifesto; 2001.
URL: http://http://www.agilemanifesto.org
Accessed 19 April 2011.

6 Fewster M, Graham D. Software test automation: effective use of test execution tools. Harlow, England: Addison Wesley Longman Ltd; 1999

7 Martin RC. Professionalism and test-driven development. IEEE Software 2007;24(3):32-36

8 Jeffries R, Melnik G. TDD: the art of fearless programming. IEEE Software 2007;24(3):24-30

9 Duval PM, Matyas S, Glover A. Continuous integration: improving software quality and reducing risk. Crawsfordsville, Indiana: Pearson Education; 2007.

10 Balch M. Complete digital design: a comprehensive guide to digital electronics and computer system architecture. USA: McGraw-Hill; 2003.

11 Decaluwe J. MyHDL manual: release 0.7 [online]. Jan Decaluwe; 2010.
URL: http://www.myhdl.org/lib/exe/fetch.php/doc:myhdl.pdf
Accessed 25 April 2011

12 Lutz M. Learning Python. 3rd Edition. Sebastopol, CA: O'Reilly; 2008.

13 Altera Corporation. Cyclone II device handbook, volume 1 [online]. Altera Corporation; 2008.
URL: http://www.altera.com/literature/hb/cyc2/cyc2_cii5v1.pdf
Accessed 27 April 2011

14 Microchip Technology. ENC424J600/624J600 Data sheet: stand-alone 10/100 Ethernet controller with SPI or parallel interface [online]. Microchip Technology; 2009
URL: http://ww1.microchip.com/downloads/en/DeviceDoc/39935b.pdf
Accessed 1 May 2011

15 Altera Corporation. Quartus II Web Edition software [online]. Altera Corporation; 2011
URL: http://www.altera.com/products/software/quartus-ii/web-edition/qts-we-index.html
Accessed 3 May 2011

16 Altera Corporation. Quartus II handbook version 10.1 volume 1: design and synthesis [online]. Altera Corporation; 2010
URL: http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf
Accessed 3 May 2011

17 Decaluwe J. Stopwatch [online]. Jan Decaluwe; 2006.
URL: http://myhdl.org/doku.php/cookbook:stopwatch
Accessed 4 May 2011

18 Robot Framework. User Guide [online]. Robot Framework; 2011
URL: http://robotframework.googlecode.com/hg/doc/userguide/
RobotFrameworkUserGuide.html?r=2.5.7#introduction
Accessed 4 May 2011

19 Google Inc. Google Code [online]. Google Inc.; 2011
URL: http://code.google.com/
Accessed 6 May 2011

20 Jenkins Community. Welcome to Jenkins CI! [online]. Jenkins Community; 2011
URL: http://jenkins-ci.org/
Accessed 6 May 2011

Appendix 1. Poart hardware user guide

Poart **input power voltage** range: +6V to +20V. **Note:** The negative voltage (or ground) is connected to the tip of the connector.The PWR LED indicates that the board is on. The RST pushbutton resets the power of the board without the need to unplug it

The **onboard 32 MHz clock oscillator** is connected to FPGA pin BANK1_23 (CLK0).

Poart provides 2 100-pin **external connectors**. One of the external connectors is marked as J3, the second one, even though not marked on the PCB is referred to as 'J2'. This corresponds to the schematic component names. Table 1 and 2 present the pin map of the external ports. Pins not listed in the tables are left floating, i.e. no-connects. FPGA pins are presented in the format 'BANKx_y'.

Table 1. Extension port J2 pin map

J2 Pin	**Function**	**J2 Pin**	**Function**
1	Raw power voltage	2	Raw power voltage
3	Raw power voltage	4	Raw power voltage
5	GND	6	GND
7	GND	8	GND
9	+3.3V	10	+3.3V
11	+3.3V	12	+3.3V
13	BANK2_160	14	BANK2_185
15	BANK2_161	16	BANK2_187
17	BANK2_162	18	BANK2_188
19	BANK2_163	20	BANK2_189
21	BANK2_164	22	BANK2_191
23	BANK2_165	24	BANK2_192
25	BANK2_168	26	BANK2_193
27	BANK2_169	28	BANK2_195
29	BANK2_170	30	BANK2_197
31	BANK2_171	32	BANK2_198
33	BANK2_173	34	BANK2_199

35	BANK2_175	36	BANK2_200
37	BANK2_176	38	BANK2_201
39	BANK2_179	40	BANK2_203
41	BANK2_180	42	BANK2_205
43	BANK2_181	44	BANK2_206
45	BANK2_182	46	BANK2_207
47	BANK1_30	48	BANK2_208
49	BANK1_31	50	BANK1_3
51	BANK1_33	52	BANK1_4
53	BANK1_34	54	BANK1_5
55	BANK1_35	56	BANK1_6
57	BANK1_37	58	BANK1_8
59	BANK1_39	60	BANK1_10
61	BANK1_40	62	BANK1_11
63	BANK1_41	64	BANK1_12
65	BANK1_43	66	BANK1_13
67	BANK1_44	68	BANK1_14
69	BANK1_45	70	BANK1_15
71	BANK1_46	72	BANK1_24
73	BANK1_47	74	BANK1_27
75	BANK1_48	76	BANK1_28

Table 2. Extension port J3 pin map

J3 Pin	**Function**	**J3 Pin**	**Function**
1	Raw power voltage	2	Raw power voltage
3	Raw power voltage	4	Raw power voltage
5	GND	6	GND
7	GND	8	GND
9	+3.3V	10	+3.3V
11	+3.3V	12	+3.3V
13	BANK4_56	14	BANK4_81
15	BANK4_57	16	BANK4_82
17	BANK4_58	18	BANK4_84

19	BANK4_59	20	BANK4_86
21	BANK4_60	22	BANK4_87
23	BANK4_61	24	BANK4_88
25	BANK4_63	26	BANK4_89
27	BANK4_64	28	BANK4_90
29	BANK4_67	30	BANK4_92
31	BANK4_68	32	BANK4_94
33	BANK4_69	34	BANK4_95
35	BANK4_70	36	BANK4_96
37	BANK4_72	38	BANK4_97
39	BANK4_74	40	BANK4_99
41	BANK4_75	42	BANK4_101
43	BANK4_76	44	BANK4_102
45	BANK4_77	46	BANK4_103
47	BANK4_80	48	BANK4_104
49	BANK3_133	50	BANK3_105
51	BANK3_134	52	BANK3_106
53	BANK3_135	54	BANK3_107
55	BANK3_137	56	BANK3_108
57	BANK3_138	58	BANK3_110
59	BANK3_139	60	BANK3_112
61	BANK3_141	62	BANK3_113
63	BANK3_142	64	BANK3_114
65	BANK3_143	66	BANK3_115
67	BANK3_144	68	BANK3_116
69	BANK3_145	70	BANK3_117
71	BANK3_146	72	BANK3_118
73	BANK3_147	74	BANK3_127
75	BANK3_149	76	BANK3_128
77	BANK3_150	78	BANK3_129
79	BANK3_151	80	BANK3_130
81	BANK3_152	82	BANK3_131
83		84	BANK3_132

Appendix 2. Extension board hardware user guide

The Ethernet extension board is designed to work with Poart's J3 extension port. The POWER LED on the extension board is lit when the board is powered.

For debug purposes the board provides two user programmable LEDs (L1 and L2) and a pushbutton (PB).

Table 3. User programmable LEDs and PB pin mapping to Poart's FPGA pins

L1	BANK3_147
L2	BANK3_146
PB	BANK3_149

The following table presents the connections of the Ethernet controller to the FPGA.

Table 4. Ethernet controller connections to the Poart FPGA

Controller pin	Function	FPGA pin
5	AD4	BANK4_68
6	AD5	BANK4_69
7	AD6	BANK4_70
8	AD7	BANK4_72
9	A5	BANK4_74
10	A6	BANK4_75
11	A7	BANK4_76
12	A8	BANK4_77
13	A9	BANK4_80
19	A10	BANK3_105
20	A11	BANK4_104
33	CLKOUT	BANK3_129
34	INT/SPISEL	BANK4_103
35	AD8	BANK4_102
36	AD9	BANK4_101
37	AD10	BANK4_99

38	AD11	BANK4_97
39	AD12	BANK4_96
40	AD13	BANK4_95
41	AD14	BANK4_94
42	AD15	BANK4_92
43	A12	BANK4_90
44	A13	BANK4_89
45	A14/PSPCFG1	BANK4_88
48	WRH/B1SEL	BANK4_87
49	CS	BANK4_86
50	SO/WR/WRL/EN/B0SEL	BANK4_84
51	SI/RD/RW	BANK4_82
52	SCK/AL/PSPCFG4	BANK4_81
53	AD0	BANK4_56
54	AD1	BANK4_57
55	AD2	BANK4_58
56	AD3	BANK4_59
57	A0	BANK4_60
58	A1	BANK4_61
59	A2	BANK4_63
60	A3	BANK4_64
61	A4	BANK4_67

Appendix 3. MyHDL Unit test code snippet

```
def enable_signal():
    """ Test the Enable signal functionality """
    Clk, Enable, PbIn_Low, PbOut_Low = [Signal(bool(0)) for i in range(4)]
    PbIn_High, PbOut_High = [Signal(bool(1)) for i in range(2)]

    clk_gen = clk_driver(Clk)

    mut_active_high = pb_debouncer(
                               Enable
                               ,Clk
                               ,PbIn_High
                               ,PbOut_High
                               ,clkHz=100
                               ,activeLevel = ACTIVE_HIGH
                               ,samplingFrequency=100
                               ,numberOfEqualSamples=4
                              )

    mut_active_low = pb_debouncer(
                              Enable
                              ,Clk
                              ,PbIn_Low
                              ,PbOut_Low
                              ,clkHz=100
                              ,activeLevel = ACTIVE_LOW
                              ,samplingFrequency=100
                              ,numberOfEqualSamples=4
                             )

    @always(Clk.negedge)
    def monitor():
        print "Time: %s" % now()
        assert(1 == PbOut_Low)   # mut_active_low drives it high
        assert(0 == PbOut_High)  # mut_active_high drives it low

    return instances()

def test_enable_signal():
    sim = Simulation(enable_signal())
    sim.run(10) # 10 time steps
```