Informatica — Universiteit van Amsterdam

Bachelor Informatica

University of Amsterdam

A Model for Experiment Setups on FPGA Development Boards

Matthijs Bos

2016-08-17

Supervisors: A. van Inge & T. Walstra, University of Amsterdam

Signed:

Abstract

This thesis proposes a model for the application of FPGA development boards as a tool for experimentation with digital logic. This model adopts the concept of an address space as a means for isolation of board-specific components in the logic's architecture, as well as a means for generalization of experiment setup interaction. This subsequently allows for the reuse of a number of architectural components across different projects as well as for the automation of different parts of the development process. As a result, the process for development of new FPGA configurations is shortened and reduced in terms of complexity.

Contents

CHAPTER 1

Introduction

The use of field-programmable gate arrays (FPGA) has increased a lot in the past years. FPGAs have been successfully applied in numerous fields of industry as well as different areas of academic research. Manufacturers have put an effort in the further development of FPGAs, resulting in an increase in performance and size, as well as a decrease in power consumption and unit cost. These developments have put FPGAs within the reach of academic education. Nowadays, a basic but fully-featured FPGA development board can be acquired for under $50.

Among many other universities, the University of Amsterdam (UvA) considers Computer architecture and organization to be an important subject in the computer science curriculum. The subject is taught to undergraduate students as an eight week course in an early stage of their first year. The subject's body of knowledge ranges from a low level understanding of a computer's central processing unit's (CPU) inner workings, design and surrounding systems to a more abstract view of the CPU that considers its instruction set from a software point of view.

The UvA's course contents are based on the widely adopted works of Hennessy and Patterson, supported by a number of lab experiments. These lab experiments allow students to reinforce their theoretical understanding of the course's subject matter through interaction with a number of virtual computer architectures in a simulator environment. In the process of modernizing the computer architecture and organization course, the instructors sense the need for a more hand-on experience in which students are capable of interacting with physical devices. This physical view of a computer architecture may aid in a student's understanding of the subject matter.

FPGAs allow for a computer architecture design to be taken out of the virtual environment of a simulator and be implemented in a physical device. As such, they may provide a solution in achieving the goals of the UvA's computer architecture and organization course instructors. The technical capabilities of FPGAs combined with the decreased cost of FPGA development boards makes their adoption in the UvA's course a viable option.

Utilizing FPGAs in an experimental setup as well as the development of these setups are complex tasks that requires a specific set of skills, including digital design, hardware definition language (HDL) programming and familiarity with specialized development tools. The complexity of FPGAs combined with the set of skills required raises problems in the application of FPGAs in an eight week introductory level computer architecture and organization course. The development of these individual skills may deserve a course on their own and are beyond the scope of the UvA's undergraduate computer science curriculum.

This thesis proposes a generic solution that aims for FPGAs to become an effective tool in education and experimentation. The proposed model for development and interaction removes the need for a number of the aforementioned skills through abstraction. This thesis focuses on providing a technical solution and does not aim to evaluate the didactical effects of the proposed model. In implementation of the proposed model is provided as well as an evaluation of its effects on a functional and technical level.

1.1 Problem Statement and Related Work

In their teaching of computer architecture and organization at the University of Amsterdam's (UvA) department of computer science, the course instructors see an opportunity for improvement through introduction of FPGA development boards as a tool for experimentation during practical exercises. The instructors sense the need for a hands-on approach in teaching their subjects, since their students lack experience with physical hardware. The course is taught early in the first year of undergraduate students over a period of two months, so students are not expected to have any academic experience in the field of computer science.

1.1.1 A Shift in Focus

Although computer architecture and organization is part of the curricula of electrical engineering, computer engineering and computer science students, its place is different in each curriculum. For electrical engineering students, digital systems are the final station in terms of abstraction. For computer science students, it's the start of a journey. Computer engineering can be considered to be in the middle of this spectrum. The UvA's instructors aim to shift the course's focus to the abstract aspects of digital systems, such as instruction-level parallelism and caching strategies in computer architectures. The instructors find a higher-level level view of digital systems on a register-transfer level and an instruction level to be more in line with a computer science student's curriculum. Only a basic understanding of lower-level subjects such as digital design and electronics is pursued. In their decision, the instructors follow a trend that was observed in [1, p. 205]: a de-emphasis of digital design in computer science curricula. Deep coverage of digital design is considered to more appropriate in the curricula of electrical engineering and computer engineering.

The UvA's computer architecture and organization course contents are based on the works of Hennessy and Patterson [2] which considers the design process of a microprocessor from a higher-level view. In their work there is a significant focus on the numerical analysis of decisions made in that process. The instructors' intended goal for their practical sessions is to allow students to analyse the implications of these higher-level design decisions through experimentation. As also noted in [3], students who have not experimented with abstract subjects such as pipelining and caching show lower test scores in these areas.

By way of example, the instructors would like to be able to provide students with a series of implementations of MIPS architectures, corresponding to those described in [2, Ch. 4]. Students could be asked to write a program that would demonstrate the effects of some optimization that is described in the textbook. The introduction of some form of hazard detection, for instance. Students could then be required to execute their programs on both computer architectures and describe their observations as well as the measured performance effects.

1.1.2 Physical component

Moving away from the lower-level aspects of computer architecture and organization contradicts with the instructors' observation that computer science students lack in their experience with computer hardware. However, the UvA's instructors believe that the introduction of a physical component in their practical sessions could present a solution. Although the instructors aim to focus on the more abstract subjects, letting students experiment with these subjects through physical devices could implicitly develop a student's intuition for hardware-related concepts and support students in gaining a basic understanding of the lower-level aspects of computer architectures. Furthermore, the instructors believe that this approach to teaching will allow students to gain a better insight in more abstract concepts, since students are enabled to see these concepts implemented in real, functioning hardware.

The instructors have identified FPGAs as a viable candidate to fulfill this need for a physical component in their teaching. FPGAs have been adopted in teaching for similar reasons. In [4], [5] FPGAs have been included in the teaching as a solution to the need for a hands-on approach in the teaching of computer architecture and organization. In [6] is observed that computer science lack experience with physical hardware. FPGAs have been applied in the curricula of

[3]–[13] and have proven to be a valuable tool in teaching computer architecture and organization. In [11] students have reported an increased insight into the subjects taught. In particular, an increased insight in abstract subjects such as pipelining is observed, since students are provided with an accurate view of the computer architecture's internal state. Furthermore, the adoption of FPGAs had resulted in an increase in student motivation, especially when subject became more challenging. In [5] students have reported an increased learning experience. In [10] the adoption of FPGAs has led to the observation that students gained an increased confidence with hardware and has provided them with an overall view of a computer system as well as its relation to software. Furthermore, visual verification of results on the FPGA development board has helped their students in gaining a better understanding of computer systems and an increased motivation. In [3] an increased student pass rate, student score and student evaluation were observed. In [4] students have reported an increased interest in the subject matter as well as an increased insight In [6] students have showed better performance and understanding through the adoption of FPGA development boards, as opposed to simulation-based methods.

1.1.3 Implementation-based Methods

Known methods for teaching computer architecture and organization using FPGAs, such as [3], [4], [6]–[10], [12], [13] focus on implementation and validation. These methods describe how students are enabled in creating a working computer architecture through implementation. FPGAs are mainly used as a means for students to validate their work, after having designed and simulated their work using electronic design automation (EDA) tools.

Due to the complexities involved in implementing a computer architecture, combined with a limited amount of time, students are constrained in the complexity of their implementations. Computer architecture is considered to be a challenging task, in which one has to address problems on different abstraction levels. As a result, these methods primarily feature implementation of simplified single-cycle machines which are limited in instruction sets and applied optimizations. Only in [10] a method is presented in which students focus on the implementation of a pipelined MIPS architecture. Although this method enables students in creating a working design over a period of two months, more advanced optimizations are omitted from the project due to a lack of time and no attention is paid to other subjects such as caching and I/O. Furthermore, none of these methods ask students to evaluate the decisions they have made during in the process of implementing their designs. The UvA's course instructors believe that these methods suffice in the teaching of basic computer architecture concepts, but are insufficient in explaining more complex subjects to students.

The time available to students for implementation is limited by the acquisition of the skills required in order to successfully implement a working computer architecture on a FPGA. Not only are students required to have an understanding of digital design and digital systems, but familiarity with additional subjects such as HDL modelling, FPGA development and experience with the involved EDA tools is also required. As a consequence, these methods are mostly targeted at more experienced students who have studied these prerequisite subjects earlier in their programmes. The methods presented in [9], [10] are targeted at students who are in later stages of undergraduate programs. The methods presented in [7], [8], [12] are known to be targeted at graduate students.

Since the UvA's course is targeted at inexperienced first-year computer science students, the acquisition of these skills would take up a significant amount of time before they could create designs of meaningful definition. As shown in [7, Fig. 1], one could designate separate courses for each of these subjects individually. Only in [4] a method is presented in which first-year students are taught the subject of computer architecture and organization whilst simultaneously acquiring a basic understanding of HDL programming through implementation exercises. An important observation is that first-year students struggle in adopting the concurrent characteristics of HDL modelling, as opposed to the imperative style of programming they may already be familiar with.

As previously stated, the UvA's computer science students are only required to be familiar with the concepts of digital systems and to have a basic understanding of digital design. The aforementioned additional skills, such as HDL modelling are not part of the students' curriculum. The students would be unnecessarily burdened by these requirements. One could conclude that

the discussed implementation-based methods are thus better suited to the curricula of electrical engineering and computer engineering, since the prerequisite subjects are often taught as part of these curricula.

1.1.4 Ready-made Solutions

Not all known methods that include FPGAs in their teaching require students to create their own implementations. The methods described in [11], [14], [15] consider FPGAs as a tool for containment of existing computer architectures. Through these methods, students are enabled to execute their programs on 'real hardware' and interact with the computer architecture's internal state through software on the user's PC. As opposed to ASIC-based methods that serve a similar goal, these methods have the advantage of providing students with a detailed view of the computer architecture's internal signals and state.

Unfortunately, these methods do not allow the UvA's instructors to provide students with a series of varying computer architecture implementations. Only limited variation is offered in the implementations of these computer architectures. In [11], [14], a single-cycle version and a pipelined version of these computer architectures are offered. In [15] only a single computer architecture is provided, although extensive in terms of features. These methods' primary focus is to enable students to experiment with computer architectures on an instruction level and provide insight into the computer architecture's internal state.

In [11], [14] specialized logic is developed for containment of the computer architecture into the FPGA and for communication with the PC software. As also noted in [11], it is designed for a specific FPGA development board and not very portable due to the use of board-specific features. This dependency makes it difficult for these methods to be adapted to other environments. In [15] a more modular approach is taken in which a simple wrapper is provided for embedding the computer architecture in a number of popular FPGA development boards. An EJTAG module is developed that serves as the primary means of interaction with the computer architecture. Through adoption of industry standards, existing hardware and software tools can be used for programming and debugging purposes. Board I/O pins expose this module's interface to the user. However, this approach prevents this method from being a self-contained solution. Separate programming and debugging hardware is required in order for students to be able to interact with the computer architecture and their executing programs via PC software.

The PC software provided in [11], [14] is easily understood and little knowledge other than that of the computer architecture, its instruction set and programming is required in order for students to be able to use it in their practical sessions. However, this software is specifically developed for a single combination of development board and its contained logic (computer architecture and encapsulating logic). These dependencies limit the use of these PC software implementations to their respective methods. Any change in development board, the computer architecture or its encapsulating logic would require modification of the PC software. The dependencies identified between these methods' components do not only limit their use, but complicates their development process as well.

Adapting these methods' implementations to fit the UvA's instructors' goals could be possible, but would require an investment that is not easily repaid. These methods lack a generic approach to the problem of embedding a computer architectures into FPGA development boards. Applying these methods' approaches to a large number of varying computer architectures would produce a solution that is not easily maintained, due to the dependencies that are created in the process of development. Furthermore, the final solution would require extensive modifications to be adapted to other environments, due to the limitations of a board-specific development process. Additionally, such an approach would distract instructors from their task of developing computer architectures for their education and would require them to focus on peripheral matters.

1.1.5 Wrap-up

The UvA's computer architecture and organization course instructors require a solution that allows their students to experiment with a series of varying computer architecture implementations. FPGAs may provide a physical approach to this problem, but current methods incorpo-

rating FPGA development boards do not provide a satisfying solution. Implementation-based methods do not manage to address the more complex subjects and are not suited to curriculum of the UvA's computer science students, due to the skills required. Students must be able to use FPGAs solely as tool for containment of ready-made computer architectures, without being burdened with the complexities of these devices. Hiding these complexities allows for students to focus on the higher-level aspects of these computer architectures and performing meaningful experiments, while the physical approach aids students in gaining an intuition for computer hardware.

Some existing methods hide these complexities form their students, but only feature static computer architecture implementations. These systems are built from tightly-coupled components that do not allow for easy modification of the featured computer architecture. In order for the UvA's instructors to achieve their goal, a generic approach to embedding digital systems in FPGAs is required, allowing instructors to provide their students with a range of varying computer architecture implementations, while reusing standard components for containment within the FPGA and using generic PC software. The instructors' work should focus on implementing a computer architecture. Furthermore, their work should be easily adaptable to different FPGA development boards.

1.2 Thesis Scope

Although this thesis' problem statement discusses various didactic aspects of the teaching of computer architecture and organization, evaluation of the proposed solution's educational value is explicitly excluded from this thesis' scope. The goal of this thesis is to find a technical solution that satisfies the UvA's instructors' requirements as described in the problem statement.

Achieving platform independence is not a goal of this thesis as well, but has been taken into account and the model developed in this thesis offers handles in achieving this. It is discussed in section 6.2 which discusses possible future work.

1.3 Research Questions

Considering this thesis' problem statement and scope, two research questions can be derived. The first research question considers the process of developing an experiment setup that is contained within a FPGA development board:

> How can the process of FPGA experiment setup development be altered such that it allows for reuse of loosely-coupled components and separation of concerns?

This thesis' second research question focuses on the process of experimentation and interaction with experiment setups contained within a FPGA.

> How can FPGAs be applied in computer architecture and organization lab experiments, while hiding their technical complexities and removing the need for HDL programming skills?

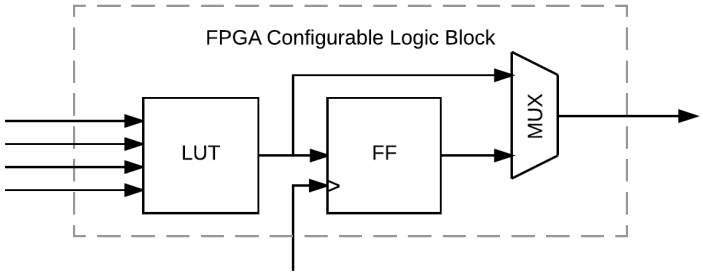
CHAPTER 2

Background

2.1 Field-programmable gate arrays

Field-programmable gate arrays (FPGA) are a class of integrated circuits (IC) which' behaviour is reconfigurable, as opposed to application specific integrated circuits (ASIC). In general, FPGAs consist of blocks of configurable logic which are interconnected through a configurable network. While some of a FPGA's physical pins are reserved, a majority of a FPGA's pins are generally available to user I/O and are exposed internally through I/O blocks (IOB). Manufacturers combine configurable logic blocks (CLB) with additional static functionality ranging from primitives such as memory elements to complex functionality such as embedded microprocessors. The included static functionality differs per FPGA model and manufacturer and many different variations currently exist, targeted at various applications.

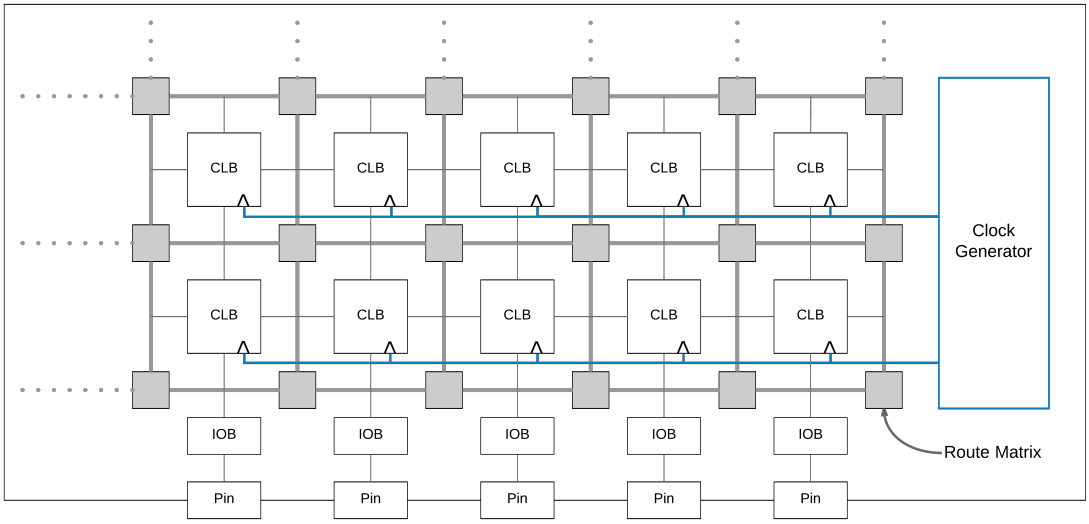
Figure 2.1: A basic view of a FPGA configurable logic block, derived from [16]. The block supports synchronous operation through the integrated flip-flop (FF).



In its most primitive form, a FPGA CLB can be described as a lookup table (LUT) whose output is optionally buffered by a flip-flop, as displayed in figure 2.1. The LUT can be configured to capture combinational logic and the flip-flip is added in order to support synchronous operation such that the CLB can be configured to capture synchronous sequential logic. This simplistic view does not represent the CLBs as used in the designs of FPGA manufacturers, however. As can be seen in [17], a Xilinx 7 series FPGA's CLBs are significantly more complex. Manufacturers include more functionality in CLB designs such that these can be used more efficiently to serve as memory elements or perform arithmetic operations, for example.

In order to combine multiple blocks of logic, CLB inputs and outputs need to be interconnected. FPGA manufacturers adopt different topologies in designing their CLB interconnect networks. Although different in structure, these networks share the capability of being reconfigurable such that routes may be defined between different blocks' input and output ports. Figure 2.2 displays a FPGA architecture in which the CLBs and their interconnects are aligned in a

Figure 2.2: A basic view of a FPGA's architecture, derived from [18, Fig.2]. CLBs are aligned in a symmetrical array. The gray boxes and interconnecting lines represent the FPGA's configurable interconnects. The blue lines represent the FPGA's clock delivery network. I/O blocks (IOB) serve as entry points to FPGA pins.



symmetrical array structure. As described in [18, Fig.2], other topologies can be adopted as well. Row-based and hierarchical structures for example, are other types of structures adopted by manufacturers in their FPGA designs.

2.2 FPGA Clocking Resources

In order for CLBs to be configured as elements of synchronous sequential logic, the CLB requires a clock input signal. Distributing this clock signal through the configurable CLB interconnect network however, causes an unpredictable amount of clock skew accross different CLB clock inputs. In order to overcome this unpredictability, FPGAs are generally equipped with a dedicated clock delivery network. The blue lines in figure 2.2 represent such a clock delivery network. Manufacturers adopt different design strategies in order to minimize clock skew accross their FPGAs. As can be seen in [19] for example, the Altera Cyclone V class of FPGAs feature a clock delivery network that is defined hierarchically.

Additional to dedicated logic for clock delivery, current FPGAs generally offer some configurable functionality for synthesis, selection and manipulation of clock signals. Several independent clock signals of different frequencies can be derived from a base signal through multiplication and routed to different sections on the FPGA. This base signal is usually provided by an external clock generating device. Besides clock synthesis and routing, FPGAs generally feature buffers that allows for 'gating' of clock signals through which a specific clock signal may be (regionally) disabled through user-defined logic. The technique of clock gating is generally used to disable state transitions in a particular section of an IC in order reduce power consumption. In this thesis however, this technique will be utilized to generate clock pulses with irregular intervals.

2.3 FPGA Development Process

Developers generally define their designs' logic on a register-transfer level (RTL) through some hardware definition language (HDL), such as Verilog or VHDL. In order for these designs to be translated into a FPGA configuration, these designs are first processed by a logic synthesizer. The goal of the logic synthesis process is to translate the hierarchical RTL logic representation

into a 'flat' netlist representation which is composed of elements that correspond with FPGA primitives.

In order to translate the primitive netlist representation into a FPGA configuration, the design is processed in the implementation process. During this process, the netlist primitives are first mapped onto FPGA elements, but not assigned to a location yet. Users are generally capable of controlling the implementation process through specification of constraints, such as specific timing requirements. After succesfully mapping, the exact location has to be determined such that routes between the elements' inputs and outputs can be established while satisfying the user-defined constraints. This process of finding a suitable combination of locations and routes is generally the most time-consuming of all operations. Once successful, the configuration is written to a FPGA-specific binary format such that is suitable for programming the FPGA. This type of file is commonly referred to as a bitstream file.

Although some third-party synthesis tools are available, the implementation process is generally facilitated by proprietary tools supplied by the FPGA's manufacturer, since many of the implementation process' details are based on proprietary information. Manufacturers generally bundle their development tools into a software development kit (SDK) which includes an integrated development environement (IDE) for interacion with these tools through a graphical interface. Even though the graphical interface is often a developer's primary means of interaction, scripted operation of these development tools is possible as well. The development tools developed as part of this thesis focus on scripted operation of the FPGA development process.

2.4 FPGA Development Boards

FPGAs are used in a wide range of products in which they are statically configured to perform specific tasks. There is a class of products however, that preserves the FPGA's reconfigurable behaviour and exposes this functionality directly to its end users. These products are typically used for development of new logic configurations and essentially serve as a tool during the development process. A FPGA development board generally includes a number of I/O devices for human interaction, sensory input or machine-to-machine communication. These devices range from simple LEDs and switches to external ICs with advanced functionalities. Furthermore, FPGA development board are generally equipped with pins for general purpose I/O with external devices. FPGA development boards range from boards with simple, general-purpose peripheral devices from boards with high-end components targeted at specific goals, such as digital signal processing.

One class of I/O devices that will prove of particular relevance to this thesis are the devices used for board-to-PC communication. FPGA development boards generally include some means to support the establishment of a serial point-to-point communication channel. The board generally features the electronics that allow physical compatibility with the communication medium, while the remaining higher-level logic such as an UART, is implemented in the user-defined logic contained within the FPGA. Traditionally, board-to-PC communication was often established through a RS-232 connection, but current development boards generally allow for establishing a virtual point-to-point communication channel over USB through a USB-UART bridge. Additional to a USB interface, modern boards are often equipped with an Ethernet interface as well, allowing for networked communication.

2.5 FPGA Configuration

FPGAs generally reserve a number of pins for the purpose of configuration. The terms programming and configuration may be used interchangeably. Current FPGAs generally make use of the IEEE 1532 standard for in-system programming, which extends the IEEE 1149.1 'JTAG' boundary scan protocol. Although some boards require third-party programming hardware, current FPGA development boards often include some form of programming circuitry that allows for reconfiguration of the FPGA through the user's PC.

Although the FPGA's programming interface may often be considered universal, the PC software required for programming is generally targeted at specific proramming hardware. However, FPGA manufacturers often bundle programming drivers for their and their third party partners'

boards in their SDKs. Additionally, FPGA manufacturers such as Xilinx and Altera provide smaller 'lab' editions of their SDKs that only include the tools and drivers required for configuration. Through these programming tools, the user is capable of uploading a bitstream file that was previously created during the development process.

CHAPTER 3

Model

This chapter will propose a model that provides a theoretical solution to the problem statement as described in section 1.1. This model addresses the problem statement from a technical and architectural point of view, as well as how user processes are affected. As a starting point, a description of a basic model for experiment setups on FPGA development boards is given in section 3.1, which is then further developed through a series of stages. In every stage, a particular aspect of the problem statement is addressed, forming a complete solution eventually.

In section 3.2 the possible limiting factor of the number of I/O devices on the FPGA development board is addressed by introducing the concept of a controller. This controller provides the experiment setup logic with a variable amount of virtual I/O channels. The levels of these I/O channels can then observed and controlled through the controller's interface that is exposed through a PC connection. In section 3.3 the controller is extended to allow for cycle-accurate control of experiment setups defined through synchronous sequential logic by making use of a FPGA's internal clocking resources. This new functionality is translated into new controller commands which are exposed over the PC connection. These two stages primarily focus on providing sufficient functionality for experimentation, while a number of the model's modifications are inspired by related work.

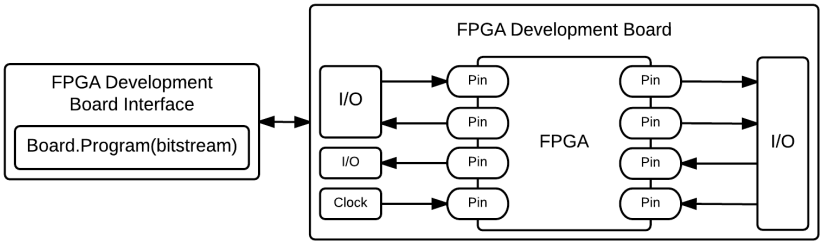
In section 3.4, the concept of an address space is proposed as a means for generalization of the interface between the controller and the experiment setup. This new level of abstraction allows for independent development of controller logic and experiment setup logic, since these components have no longer got any specific dependencies between them. This generalization is then utilized in modification of the experiment setup development process, such that it allows for reuse of components, thus reducing the amount work required for the development of new experiment setups.

In section 3.5, specific characteristics of this new experiment setup development process are exploited to allow for the introduction of new tools in that automate parts of the development process. These new tools do not only further reduce the amount of work required during development, but provide a new level of abstraction as well. Through this new level of abstraction, a part of the development process' complexity is hidden from the experiment setup developer.

3.1 The Basic Model

As a starting point, a description of a basic model for experiment setups on FPGA development boards is given. Essentially, this basic model describes how one would develop and interact with an experiment setup embedded in a FPGA using conventional methods. An overview of the basic model is displayed in figure 3.1. The model features a PC and a FPGA development board as the two primary physical components. An interface between the PC and the FPGA development board exists over which the board exposes the `Board.Program()` operation. This operation initializes the FPGA by loading the contents of a FPGA-specific bitstream file and configuring the FPGA's components.

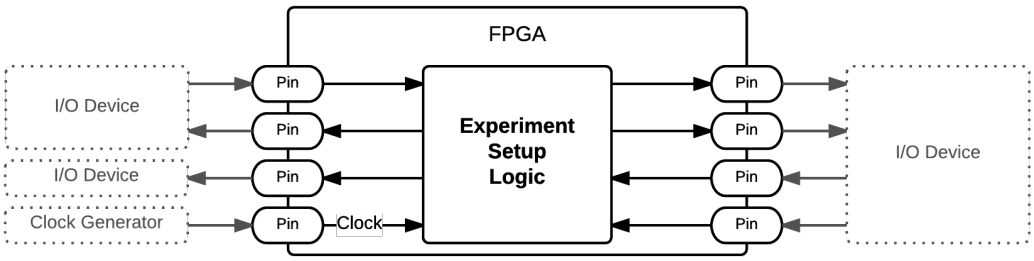
Figure 3.1: The basic model, an overview of the FPGA development board and its exposed interfaces.



In this basic model, the FPGA development board is considered to be a hosts for a FPGA and its various peripheral components. Not all pins of the FPGA's physical package and not all peripheral devices are considered to be relevant to the model. Only the pins whose signals can be controlled through the FPGA's contained logic are included. Peripheral devices that do not connect to these pins, such as power supplies or programming circuits are excluded from the model. Specifically, the presence of a clock generating device is assumed, providing the FPGA with a clock signal on one of its pins.

Besides the omission of details of the FPGA's peripherals, a part of the FPGA's internal complexities are hidden from the model as well. Figure 3.2 gives a graphical overview of the FPGA's internals and its contained logic. The FPGA's internal interface is simplified and defined to be a container for the end product of a HDL developer's work: an entity with input signals, output signals and an input clock signal. Other physical, electrical or logical characteristics of the FPGA are not included in this model.

Figure 3.2: The basic model, an overview of the FPGA and its contained experiment setup logic. No specific architecture is applied to the logic.



In the case of this basic model, no specific architecture is defined that will embed the experiment setup logic into the FPGA development board. A specific architecture will be developed in the following stages. At this stage of the model, a developer is responsible for the development of its own architecture that embeds the experiment setup logic into the environment of the FPGA.

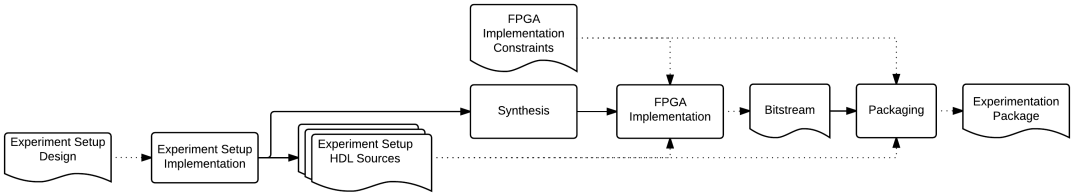
In this basic model, two user roles are defined: experiment setup developers and experimenters. In a classroom environment, an instructor can be considered an experiment setup developer and a student can be considered an experimenter.

3.1.1 Experiment Setup Development

Experiment setup developers are responsible for the design, implementation, testing, documentation and distribution of experiment setups for use on FPGA development boards. The end product of their work is a package containing a bitstream file and optionally the source files used

in the bitstream file's compilation process. This package is referred to as an experimentation package. Figure 3.3 displays a graphical overview of the experiment setup development process.

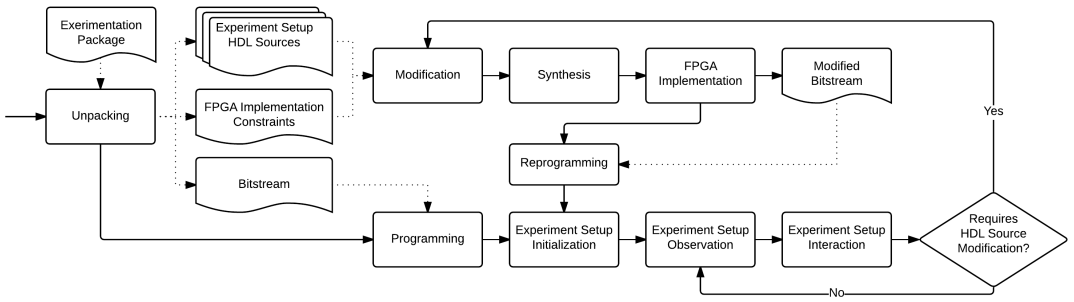
Figure 3.3: The basic model, an overview of the experiment setup development process.



3.1.2 Experimentation

Experimenters are responsible for carrying out experiments. They obtain experimentation packages and initialize experiment setups on their FPGA development boards. In order to complete the experiment, they make observations and interact with the experiment setup that is contained within the FPGA. Figure 3.4 displays a graphical overview of the experimentation process. In this basic model, two different methods of interaction are defined: interaction through board I/O devices and interaction through the process of HDL source modification, recompilation and reprogramming. Observation of the experiment's results can be done through the FPGA development board's I/O devices. Some development tools allow for live inspection of the FPGA's internal signal levels through a PC connection and specialized software.

Figure 3.4: The basic model, an overview of the experimentation process.



In order for a user to act as an experimenter, one must understand the logic and workings of the obtained experiment setup. Furthermore, an experimenter must understand the basic concepts and role of the FPGA development board, as well as how to program the FPGA via the experimenter's PC. In preparation of the experiment, experimenters must install programming software and operating system drivers for the FPGA development board. In this case, interaction and observation is done through the board's I/O devices.

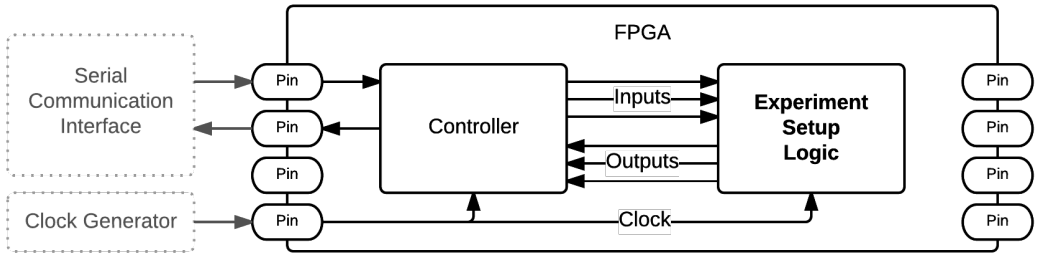
Including the PC as a tool for observation and interaction requires additional knowledge and preparation from experimenters. In order to understand and modify the experiment setup's HDL sources, users must be familiar with the programming language used. Recompilation requires users to install and understand the FPGA's development tools in order to set up a proper development environment. Furthermore, this method of interaction will significantly increase the time required for every change to be processed, since the process of recompilation is a time consuming process. Using the FPGA as a tool for observation requires further familiarization with the FPGA's development tools.

3.2 Virtualizing I/O

Regular FPGA development boards offer a limited set of I/O devices. As a consequence, this allows for observation and control of experiment setup logic entities with a limited number of input and output signals. Embedding entities with a large number of inputs and outputs however, requires a different approach.

In order to support experiment setup entities with a large number of input and output signals, the basic model's logic architecture is extended through introduction of the controller. Figure 3.5 gives an overview of the FPGA and the newly defined architecture. The experiment setup entity's inputs and output signals are available to the controller only, but both components still share a common clock signal. The signal levels of the FPGA's peripheral devices are no longer driven by the experiment setup entity. Following the previous section's criteria, these devices are considered irrelevant and thus removed from the model. Only the clock generating device and a single machine-machine communication device remain part of the model.

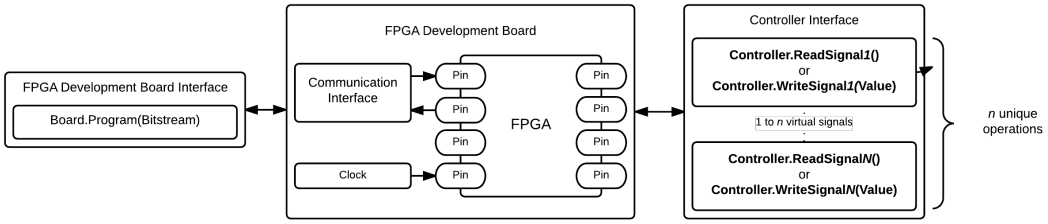
Figure 3.5: I/O virtualized, an overview of the FPGA and its contained logic architecture. The controller embeds the experiment setup logic into the FPGA.



The controller is defined to be a component that acts as an intermediary between the experiment setup logic and an experimenter's PC. The experiment setup's state is observed and controlled through controller-specific software, a method also presented in [14] and [11]. A hardware-based method to observe and control the experiment setup's state is described in [13]. A software-based solution however, allows for a flexible interface as well as a self-contained solution in terms of physical components.

The controller exposes an additional interface through a communication channel that must be provided by one of the FPGA development board's communication devices, as can be seen in figure 3.6. This interface defines operations that allow for the experiment setup entity's individual input and output signals to be controlled and observed respectively. A similar interface for observation and control of specific signals is described in [14].

Figure 3.6: I/O virtualized, an overview of the FPGA development board and its exposed interfaces.



The explicit separation of concerns in the logic architecture allows for separate implementation and validation processes for experiment setup logic, controller logic and PC software. This

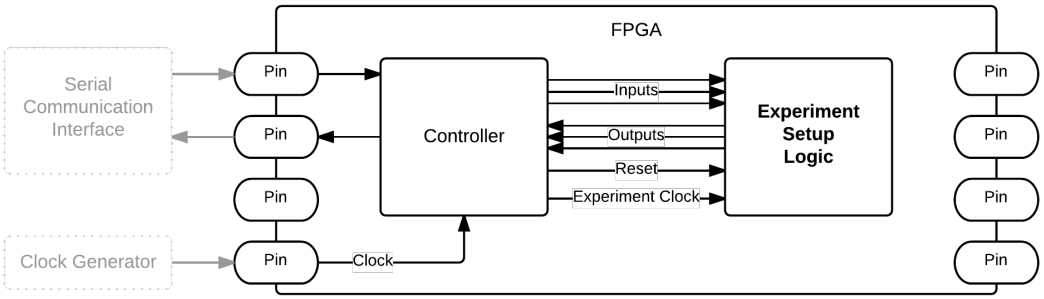
partitioning subsequently allows for distribution of work among specialists, removing the need for a single experiment developer to have knowledge and experience in all the areas previously described in section 3.1.1. As a consequence of the separation however, more dependencies are introduced into the development process. The development of a controller component requires a definition of the experiment setup's interface and any change in this interface definition requires modification of the controller. A similar dependency exists between the controller and the PC software. A change in the experiment setup entity's interface definition will thus not only result in the need for modification of the controller, but in the need for modification of the PC software as well. These dependencies are addressed and removed from the model in section 3.4.

Experiment setup interaction through dedicated PC software will simplify the experimentation process for experiment setups with large number of input and output signals. Involving the PC removes the need for experimenters to install and familiarize themselves with the FPGA's development tools. Experimenters will be able to interact with the experiment setup through a graphical interface on the PC, removing the need need for HDL programming skills and allowing for real-time interaction, since no recompilation is required. Since the board's I/O devices have been removed at this stage of the model, there is no possibility for users to physically interact with the experiment. The PC software provides the only means of interaction with the experiment setup.

3.3 Cycle control

The availability of a clock signal in the current model does provide support for experiment setup entities that are defined through synchronous sequential logic. At this stage of the model's development however, this clock signal is constant and cannot be varied in speed or stopped temporarily. Embedding an experiment setup entity of synchronous sequential logic would result in an uncontrollable situation of continuous state changes at high speed. In order to be capable of cycle-accurate observations and control over synchronous sequential experiment setup logic, the model requires further extension.

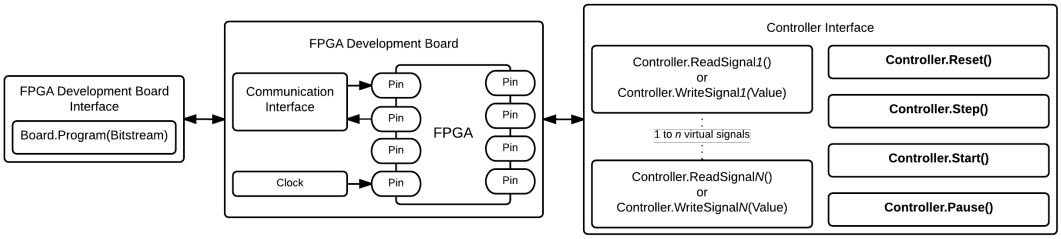
Figure 3.7: Cycle control, an overview of the FPGA and its contained logic architecture. The `Experiment Clock` and `Reset` signals are added to allow for cycle-accurate interaction and (re)initialization.



As a solution to the problem stated above, the FPGA's contained logic architecture has been extended, as displayed in figure 3.7. The most significant change is the addition of the `Experiment Clock` input signal on the experiment setup's interface definition. As described in section 2.2, a FPGA's clock delivery network generally contains buffers that allow for gating of clock signals. Utilizing this functionality in the controller's implementation will allow for the synthesis of a clock signal with pulses on irregular intervals. Combined with the `Reset` signal, these new input signals allow for cycle-accurate controlling and reinitialization of synchronous sequential logic contained within the experiment setup's logic.

In order to make this newly developed functionality available to the experimenter through the PC connection, the controller's interface is extended to support new operations that allow

Figure 3.8: Cycle control, an overview of the FPGA development board. The controller's interface is extended with operations that allow for cycle-accurate operation of the experiment setup.



for the control of cycles. As displayed in figure 3.8, four new operations are available through the controller's interface. The `Controller.Reset()` operation (re)initializes the experiment setup to it initial state and waits for the next operation. Experimenters have manual control over the experiment setup's cycles through the `Controller.Step()` operation. The controller may also be instructed to manage the experiment setup autonomously through the `Controller.Start()` operation. The `Controller.Pause()` operation stops the controller's autonomous management of the experiment setup and awaits the next operation. Similar interfaces for cycle-accurate control of experiment setups are described in [14] and [11].

3.4 Experiment Setup Interface Generalization

While the model has developed to address a number of problems encountered by experimenters, the problems experienced during development have only been partially addressed and tight coupling between logic architecture's components is still present. The source of these dependencies is the interface between controller and experiment setup logic, since every experiment setup has a unique interface. Through the establishment of a generic definition for the interface between experiment setup logic and the controller, this dependency is removed, allowing for reuse of board-specific logic in the development of different experimentation packages.

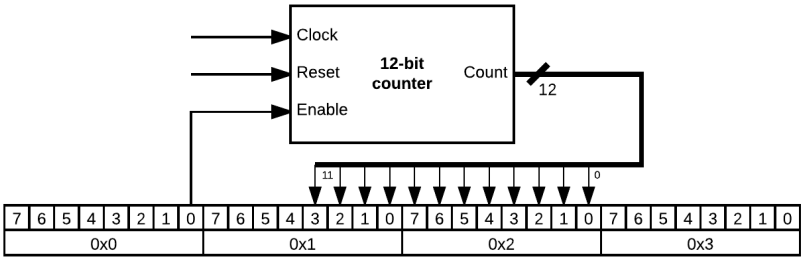
3.4.1 Address Space

This thesis proposes an address space as a means of interaction with an experiment setup's input and output signals. More specifically, an address space is defined to be a virtual array of uniquely-addressed memory slots of equal size, a model equal to that of a computer's memory. The address space features the `read(address)` and `write(address, value)` operations, allowing for interaction. The model of an address space was chosen due to its compatibility with the problem of providing a generic approach to interaction with a statefull element, as well as computer scientists' familiarity with this model of representing state. This abstraction removes the tight coupling that existed between the controller and the experiment setup, since the unique relation between these two components no longer exists.

In order for an address space's individual slot to correspond to specific input or output signals, the signals are projected onto the experiment setup's address space. Figure 3.9 displays an example of such a projection. As can be observed, all of the experiment setup's input and output signals are projected onto the address space, except for the `clock` and `reset` input signals, which are separately driven by the controller. Experiment setup signals whose width exceeds that of the address space's individual memory slots are spread over multiple slots, spanning multiple addresses. Every signal corresponds to at least one unique address.

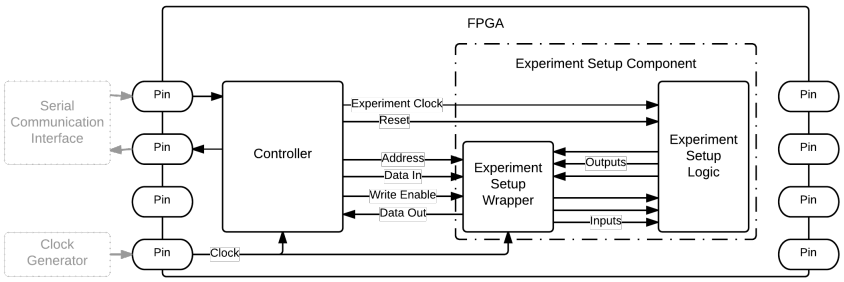
As displayed in figure 3.10, in addition to the experiment setup's primary logic, the logic architecture defines a secondary component that facilitates the projection of the experiment setup's signals on its address space. In order to allow for clear distinction between the different components, this new component is referred to as the experiment setup wrapper. The experiment

Figure 3.9: Signal projection for a 12-bit counter experiment setup. In this example, the data bus is defined to have a width of 8 bits and the address bus is defined to have a width of 2 bits, defining 4 addresses.



setup wrapper has been given its name since it "wraps" around the experiment setup logic's input and output signals. Additionally, the experiment setup's logic combined with this newly defined experiment setup wrapper is collectively referred to as the experiment setup component.

Figure 3.10: Experiment setup interface generalization, an overview of the logic architecture contained within the FPGA.



3.4.2 Logic Interface

In order to facilitate the address space abstraction, the experiment setup component's interface definition is modeled to resemble the interface of a simple block ram memory element. As displayed in figure 3.11, the experiment setup component's interface extends the block ram's interface with a `reset` input signal as well as an additional clock signal labeled `experiment clock`. This clock signal was introduced in the previous section for cycle-accurate control over the experiment setup logic. The second clock input labeled `clock` is a clock input whose signal is equal to that of the controller's. This clock signal allows for the operation of synchronous logic facilitating the address space projection.

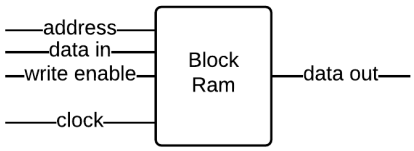
Additional to the similarities in interface signature, the experiment setup's interface is defined to have a behaviour similar to that of a block ram as well. Figure 3.12 displays a timing diagram that describes the experiment setup's signal behaviour for the `read()` and `write()` operations. Both operations are defined to take effect on rising clock edges, thus requiring the `data out` signal's levels to be buffered.

3.4.3 Generic Controller Interface

A major advantage of the generic interface between the controller and the experiment setup component is that this can be directly translated into a generalization of the controller's command interface as well. As displayed in figure 3.13, the controller interface as exposed through the PC

Figure 3.11: A comparison between the interfaces of a block ram and the experiment setup component.

(a) Single-port block ram, derived from [20, Ch.1].



(b) Experiment setup component interface.

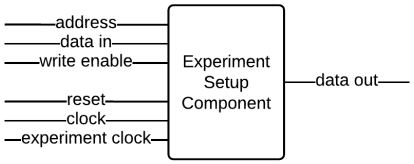
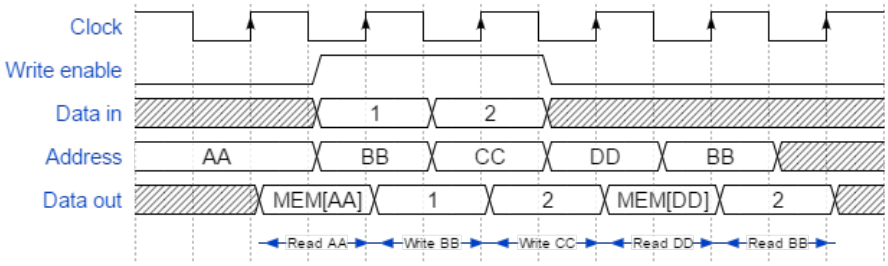
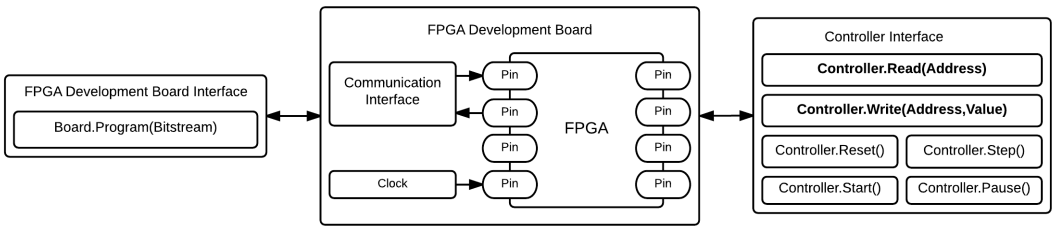


Figure 3.12: Timing diagram for experiment setup logic, derived from [20, Fig.1-2]



connection is modified such that the operations for signal interactions have been replaced by the `Controller.Read()` and `Controller.Write()` operations for interaction with the experiment setup component's address space.

Figure 3.13: Experiment setup interface generalization, an overview of the FPGA development board. The controller's interface no longer features direct control over specific experiment setup signal levels, but now features operations for interaction with the experiment setup's address space.



Since the address space itself does not describe the experiment setup logic signals' projection, additional information is required for sensible interaction with the experiment setup component. The concept of an address space description is introduced, providing information about how the experiment setup's signals are projected on the experiment setup component's address space.

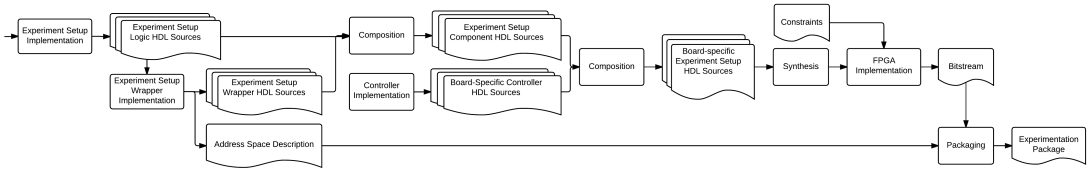
3.4.4 Processes

Figure 3.14 displays an overview of the process for the development of new experimentation packages. The experiment setup logic's HDL sources as well as the experiment setup wrapper's HDL sources are developed and composed to collectively define the experiment setup component's HDL sources. As part of the development of the experiment setup's wrapper, a description of the experiment setup's address space is created as well.

The generalization of the experiment setup component interface allows for modification of the development process. It allows for independent development of board-specific controller logic

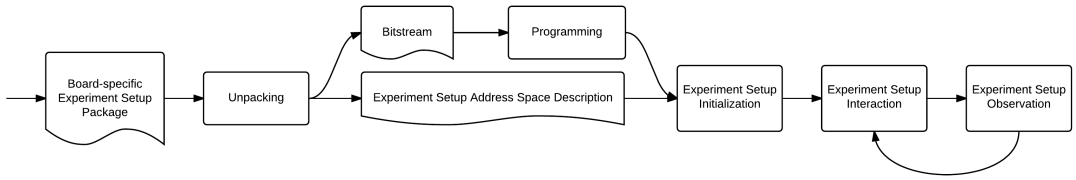
HDL sources. The definition of a generic interface allows thus for these sources to be reused in the development of different experimentation packages. The comtroller's sources are combined with the experiment setup component's HDL sources in order to define the experimentation package's complete set of HDL sources. These sources can then be taken through the FPGA's compilation in order to generate a bitstream file. This bitstream file is then packaged with an description of the address space, in order to form an experimentation package.

Figure 3.14: Experiment setup interface generalization, an overview of the experimentation package development process. The process allows for the reuse of controller HDL sources in the development of different experimentation packages.



The experimentation process as displayed in figure 3.15 is extended to include the experiment setup's address space description in the initialization of the exeriment setup, in order to allow for meaningful interpretation of the experiment setup's address space.

Figure 3.15: Experiment setup interface generalization, an overview of the experimentation process.



3.5 Development Process Automation

Due to the previous section's modifications, the experimentation package development process has become more complex. In order to reduce the process' complexity, this thesis' model definition is extended with two new abstract operations that combine a significant part of the process' operations. Additional to reducing the experimentation package development process' complexity, these new abstract operations allow be executed in an automated fashion.

As displayed in figure 3.16, the experimentation package development process has been divided into three subsequent subprocesses: experiment setup logic development, experiment setup wrapping and the component composition. The detailed contents of these subprocesses are displayed in figure3.17.

Figure 3.16: A high-level overview of the experimentation package development process. The process has been divided into three subprocesses in order to allow for automation.

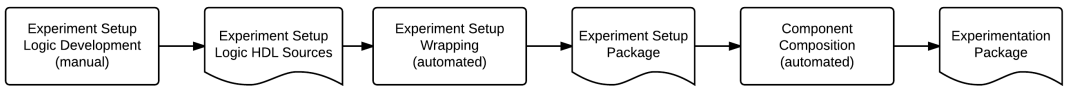
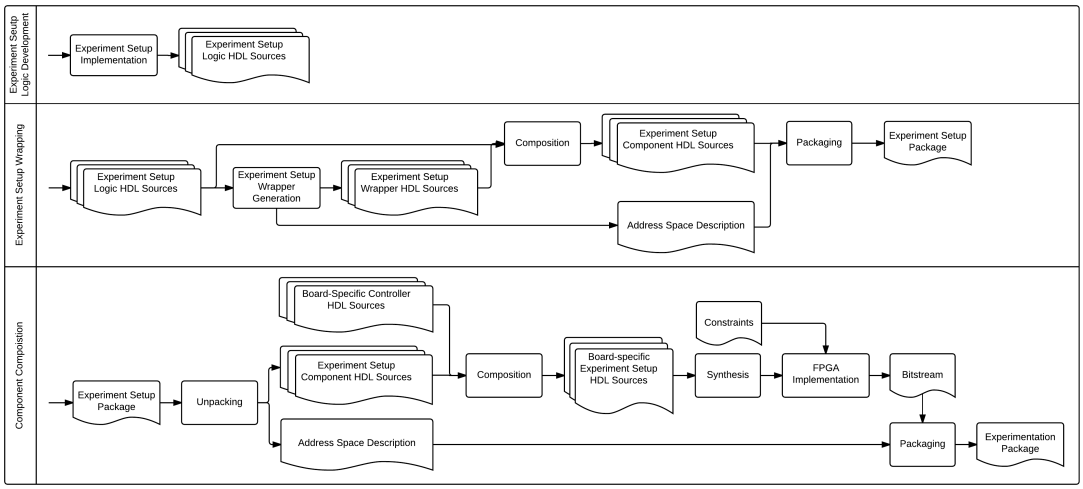


Figure 3.17: A detailed overview of the experimentation package development process' subprocesses.



3.5.1 Component Composition

Since the board-specific controller sources can now be considered static across the development processes of different experimentation pacakges, a large part of the experimentation package development process' is composed of actions that are similar accross different projects.

As an interface to this subprocess, the concept of an experiment setup package is introduced into the model's definition. This package acts as a unit of containment for the experiment setup component's HDL sources that was produced by the experiment setup wrapping process. Additional to the experiment setup's HDL sources, the package contains a description of the experiment setup's address space projection. This subprocess combines the user-defined sources from the input package with controller sources and processes these through the FPGA's toolchain. The resulting bitstream file is then pacakged into an experimentation package, together with the provided address space description.

The introduction of the component composition subprocess significantly reduces the complexity of the experimentation package development process. Due to the automated execution of its actions, these can be reduced to the abstract concept of component composition. The process abstracts the actions that relate to the FPGA and its toolchain, thus relieving the developer from the requirement of familiarizing itself with these tools and the related concepts.

3.5.2 Experiment Setup Wrapping

The generalized interface between the experiment setup component's logic and the controller require users to adapt the experiment setup logic's interface through address space projection. This process of wrapping an experiment setup's logic may be automated through source code analysis and source code generation. The potential for automated execution of this subprocess further reduces the complexity of the experimentation package's development process. Through the abstraction of this subprocess' operations, the adaption of adapting experiment setup logic to match the model's generic interface is reduced to a single operation.

The experiment setup wrapping process as well as the component composition package are defined to be two separate processes. Although these subsequent processes may be combined into one automated operation, they have been defined as separate. In order to allow for advanced address space projections or wrapping logic that cannot be generated automatically, these processes have been defined as separate.

CHAPTER 4

Implementation

In order to support the proof of concept for the model as proposed in chapter 3, a number of the model's parts have been implemented:

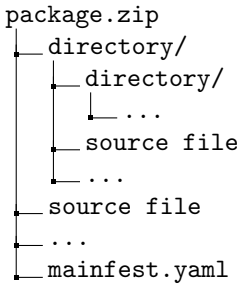
- Experiment setup package specification
- Experiment setup wrapper
- Digilent Nexys 4 controller
- Experimentation package specification
- Digilent Nexys 4 composer

As discussed while establishing the thesis' scope in section 1.2, this thesis goal is not to achieve platform independence. As such, the implementation is specifically targeted at the Digilent Nexys 4 development board and the toolchain offered by the Xilinx Vivado HL WebPack edition, version 2016.1. The possibilities of achieving platform indepence are discussed in section 6.2.

4.1 Experiment Setup Package Specification

A format for the model's experiment setup package has been defined. For the purposes of portability, a `.zip` archive is used as a means for containment of all associated files. A manifest file describes the contents of the package. This approach to packaging was derived from other packaging methods, such as java `.jar` packages. Figure 4.1 displays an overview of the experiment setup package file structure. The name and location of the manifest file is constrained by the specification, but the names and structure of other files contained in the package are not limited by any constraints.

Figure 4.1: An overview of the experiment setup package file structure



The manifest file describes the contents of the experiment setup package and its information is encoded in `YAML`[1] syntax, a widespread format for capturing configuration information. Besides

[1] http://yaml.org/

package metadata, such as title and author information, the manifest also contains file pointers and a description of the experiment setup's address space. Furthermore, the manifest defines the widths for the address and data buses. The contents of an example experiment setup package manifest file are displayed in listing 4.1. In automated processes, this manifest file is generated by the experiment setup wrapper, as described in section 4.2.

Listing 4.1: Example experiment setup package `manifest.yaml`

```
title: Full Adder using Half
description: >
    An implementation of a full adder using two half adders.
author: Matthijs Bos
date: 2016-08-17
url: https://github.com/matthijsbos/fulladderhalf/
vhdlFfiles:
    - wrapper.vhd
    - fulladder.vhd
    - halfadder.vhd
topLevelFile: ./wrapper.vhd
addressWidth: 3
dataWidth: 1
addressSpacePartitioning:
    - { name: a,    direction: in,  address: 0x0, width: 1 }
    - { name: b,    direction: in,  address: 0x1, width: 1 }
    - { name: cin,  direction: in,  address: 0x2, width: 1 }
    - { name: s,    direction: out, address: 0x3, width: 1 }
    - { name: cout, direction: out, address: 0x4, width: 1 }
```

4.2 Experiment Setup Wrapper

A tool has been developed to automate the model's experiment setup wrapping process, as described in section 3.5.2. The tool has been implemented using the Java 8 programming language and its source code is made available through a git repository[2]. This tool takes an existing VHDL implementation and a configuration file to automatically generate an experiment setup package. The tool is activated through the command line by the command `fpgaedu wrap`.

Listing 4.2 displays an example configuration file for the wrapper tool. This file contains contains pointers to the VHDL files that define the experiment setup's implementation as well as a pointer to the file that contains the implementation's top-level module. Additionally, the configuration file is also used to specify experiment setup metadata entries which are copied into the experiment setup package's manifest file.

Listing 4.2: Example `fpgaedu.yaml` wrapper configuration file for a simple 8-bit counter implementation.

```
title: 8-bit counter
description: A simple 8-bit counter with an enable signal.
url: https://github.com/matthijsbos/count8
vhdlFiles:
    - ./counter.vhd
topLevelFile: ./counter.vhd
```

A number of steps can be identified in the process of wrapping an existing experiment setup

[2] `https://github.com/matthijsbos/fpgaedu-java`

implementation. First the configuration file is read in order to obtain a pointer to the implementation's top-level file, which is then read and parsed. A VHDL parser was generated using the ANTLR4 language tool[3], for which a comprehensive VHDL grammar description was already available [4]. After parsing the top-level file, the obtained information about its input and output signals is used to define an address space projection. Based on this projection, a template is then used to generate a HDL description of the wrapper's logic.

Address Space Projection

In order to allow for signal interaction through an address space, the experiment setup's signals are to be projected on that address space. A distinction can be made between experiment setup input signals, which can be read from and written to from the address space, and experiment setup output signals, which can only be read from the address space. The address space is partitioned to provide every experiment setup signal with its own partition of this address space, such that these signals are uniquely addressable.

How signals are projected on the address space is dependent on the experiment setup package's data bus width and the widths of the individual signals. If a signal's width is smaller than the width of the data bus, there are no issues in projecting the signal on a single address. If the signal's width exceeds the data bus' width however, the signal's projection is distributed over subsequent addresses. More specifically, a big-endian approach was taken and in the case of a signal's width being less than the width of the data bus, the vacant bits on the MSB end of the address' contents are set to be zero when reading. For reasons of simplicity, every address is associated with a single signal.

Limitations

Due to the complexities of the VHDL language, support for a number of language features has been omitted. Although there is no limit on the number of ports, entity port types are limited to be of `std_logic` or big-endian `std_logic_vector`, as defined in the `IEEE.std_logic_1164` package. Furthermore, no support for generic parameters has been included in the tool's implementation. Although these limitations may prove to be problematic in production environments, their effects are not considered to be relevant in establishing a proof of concept. These types are among the most commonly used and this limitation can be easily overcome through simple modifications of the top-level VHDL file. Signals between internal instances are not subjected to these limitations.

4.3 Digilent Nexys 4 Controller

A controller has been developed for the Digilent Nexys 4 FPGA development board, displayed in figure 4.2. The board's primary component is a member of the Xilinx Artix-7 family of FPGAs. An important peripheral component is a controller that allows for the FPGA to be configured over USB, as well as the establishment of a virtual serial point-to-point connection over the same USB connection.

4.3.1 Communication Protocol

A communication protocol has been designed to facilitate communication between the PC and the Controller embedded in the FPGA. When considering the OSI model as described in [21, p.28], the protocol exists in the physical and data link layers. Furthermore, an application layer has been defined. Since the communication channel is established on a point-to-point basis and does not require any mechanisms for reliable communication, any layers in between have not been considered relevant.

[3] `http://www.antlr.org/`
[4] `https://github.com/antlr/grammars-v4`

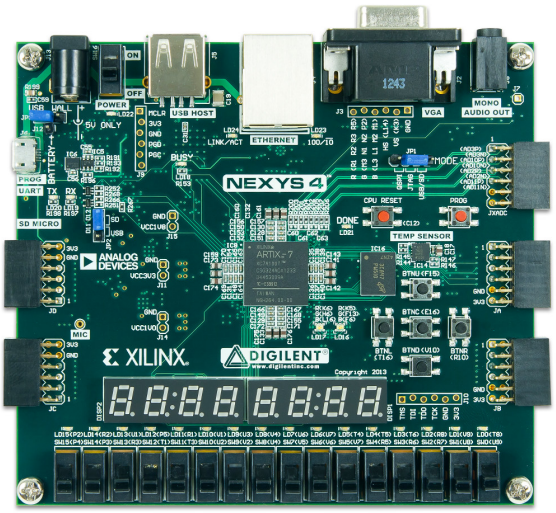


Figure 4.2: Digilent Nexys 4 Artix-7 FPGA development board

Physical Layer

On the physical layer, the FPGAs peripheral devices facilitate a large part of the communication between the controller and PC. As described in [22, p.9], the Nexys 4 board features dedicated hardware for communication with the PC through a USB-UART bridge. A Rx and Tx line are available to specific FPGA pins and a virtual "COM" ports is made available on the user's PC. As part of the logic design embedded in the FPGA, another UART[5] is used in order to receive data from and transmit data to the PC. Although the board features other means of machine-to-machine communication such as Ethernet, the approach involving UARTs was chosen for reasons of simplicity and compatibility.

Data Link Layer

UARTs typically handle byte-sized units. In order to support the transmission and reception of larger units of data, a simple data-link layer has been implemented in order to define a data frame. To minimize the complexity of the implementation, the decision was made to provide an unacknowledged and connectionless service, as described in [21, p.177]. Although a production-oriented implementation might benefit from more robustness in the data-link layer, features such as frame acknowledgement and error detection have not been considered relevant in establishing a proof of concept. Due to the current implementation's adoption of the layered OSI model however, it should be possible to modify the data link layer to support these features.

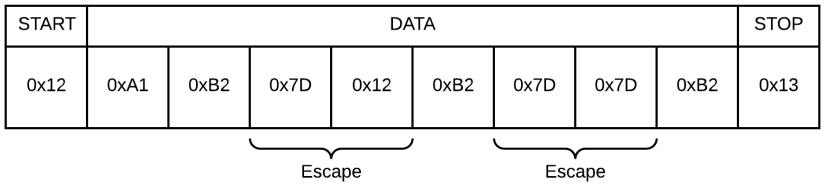
In order to identify individual frames of data, character stuffing [6] is used, as described in [21, p.180]. Specific byte values are assigned to flag a frame's start and end. An escape character is used to prevent frame data to be identified as a control character. As in [23], character `0x12` is used as a start flag, `0x13` as a stop flag and `0x7D` is used as escape character. Bytes are transmitted in big-endian order. Figure 4.3 displays an example data frame.

Application Layer

On the application layer, a message format has been defined, similar to a processor's instruction set as described in [2], for example. Messages have a fixed width and conform to be either an address-type or value-type message, as displayed in figure 4.4. Both message types reserve their most significant byte position for the message opcode, a term commonly used in instruction sets to identify an instruction's desired operation. The two message formats suffice in covering the variations that exist in the different message types. Address-type messages are used to interact

[5] Universal Asynchronous Receiver/Transmitter
[6] Also known as byte stuffing.

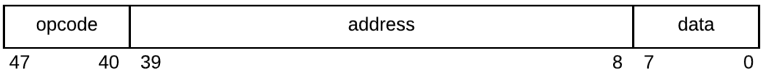
Figure 4.3: Example data frame for transmission of six bytes: `0xA1`, `0xB2`, `0x12`, `0xB2`, `0x7D`, `0xB2`. The third and fifth byte are preceded by escape character `0x7D`.



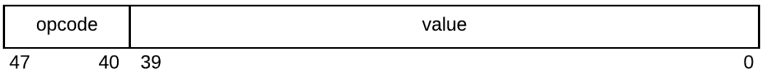
with the experiment setup's address space, while the value-type message format is used to describe the other operations. Both controller command and response messages use the same formats.

Figure 4.4: Message formats for a controller with an address width of 32 and a data width of 8.

(a) Address-type message format



(b) Value-type message format



Tables 4.1 and 4.2 list the available opcodes for controller commands and responses respectively. The command opcodes have been chosen to reflect the operations that are exposed from the controller to the PC software. As can be observed from these tables, a large number of operations do not require to pass information as arguments. One might argue that a variable-length message format could have been more appropriate. In this implementation however, the decision was made to focus on simplicity and a fixed-length approach was taken to allow for simple message processing logic in the controller.

Some controller responses have a counterpart that is defined to indicate an error in the processing of the requested command. These errors represent the error of being incapable of executing the command due to the mode in which the controller is currently operating. The controller is incapable of executing the `Step (0x3)` command for example, when operating autonomously as the result of a previous `Start (0x4)` command. A response message with opcode `Step error (0x6)` would be returned in that case.

Table 4.1: Controller commands

Name	**Opcode**	**Type**	**Arguments**
Read	0x00	Address-type	Address
Write	0x01	Address-type	Address, Data
Reset	0x02	Value-type	None
Step	0x03	Value-type	None
Start	0x04	Value-type	None
Pause	0x05	Value-type	None
Status	0x06	Value-type	None

4.3.2 Logic Architecture

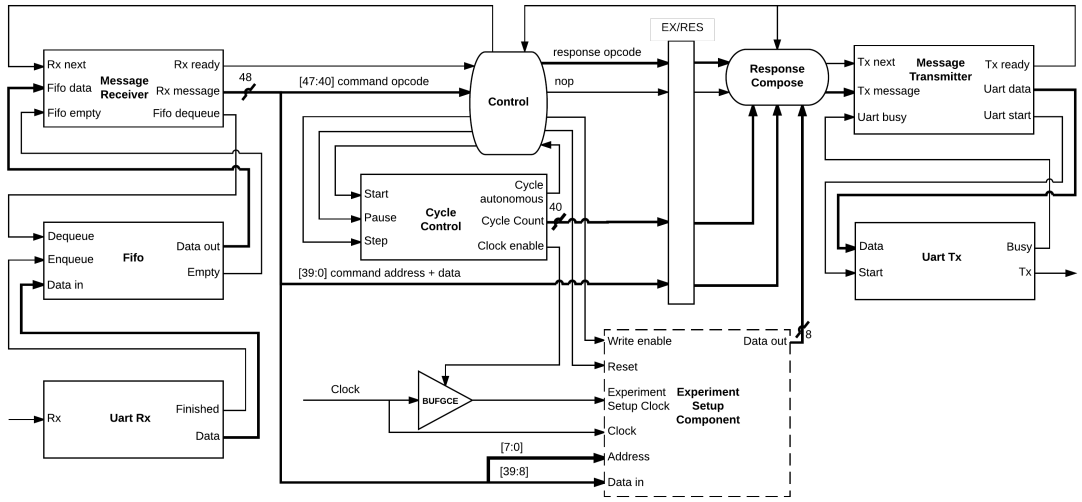
A controller for the Digilent Nexys 4 FPGA development board has been implemented. Figure 4.5 displays an overview of the controller's logic architecture. The architecture features the logic

Table 4.2: Controller responses

Name	Opcode	Type	Arguments
Read Success	0x00	Address-type	Address, Data
Read Error	0x01	Address-type	Address
Write Success	0x02	Address-type	Address, Data
Write Error	0x03	Address-type	Address, Data
Reset Success	0x04	Value-type	None
Step Success	0x05	Value-type	Value
Step error	0x06	Value-type	None
Start Success	0x07	Value-type	Value
Start Error	0x08	Value-type	None
Pause Success	0x09	Value-type	Value
Pause Error	0x0A	Value-type	None
Status Success	0x0B	Value-type	Value

for experiment setup control as well as logic that implements a communication protocol described in section 4.3.1. The controller source code is made available through a git repository [7]. The logic architecture has not been directly implemented using VHDL, but was modeled in Python code through the MyHDL[8] library. The choice for MyHDL was made to allow for increased agility during development and its capabilities for fast and efficient unit testing of components. The library allows for automatic conversion of Python sources to VHDL, so it only serves as an intermediary representation during development.

Figure 4.5: An overview of the Nexys 4 board package logic architecture, largely following the conventions for representation as defined in [2, Ch.4]. The experiment setup component's address and data buses are defined to have widths of 32 and 8 bits respectively.



PC Communication

On the physical layer, a standard UART has been implemented for both reception and transmission of bytes. The receiving UART writes to a buffer, since command messages have a length of multiple bytes. No error detection operations are performed, since it has not been considered relevant in establising a proof of concept. The data link layer has been implemented in the message receiver and message transmitter components. The message receiver component reads from

[7] https://github.com/matthijsbos/fpgaedu-nexys4-python
[8] http://www.myhdl.org/

the buffer until a valid data frame is identified through its flags. Once a complete frame has been received, the component notifies the controller of the reception of a new command message. The message is held in an internal register until the controller has signaled the receiver component to start receiving a new data frame, and thus serves as an additional buffer.

Once the controller is ready to transmit a response message to the PC, the message's contents are written to the message transmitter's internal register and the transmitter is signalled to start transmission. During transmission, the message transmitter serves as a buffer for outgoing messages. The transmitter component then sequentially instructs the transmitting UART to transmit the response message's contents, automatically inserting control flags and escaping reserved characters.

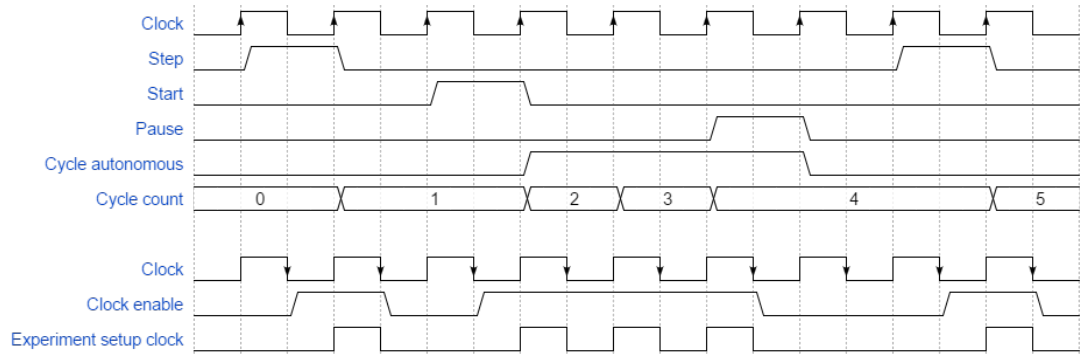
Controller

The logic for controlling the experiment setup as displayed in figure 4.5 is collectively referred to as the controller. More specifically, it concerns the blocks of combinational logic labeled `Control` and `Response Compose`, as well as the block of sequential logic labeled `Cycle Control` working in conjunction with a `BUFGCE` clock buffer. THe `BUFGCE` clocking resource is described in [24, p.40]. The components are divided over the two stages execute (`EX`) and response (`RES`), which are connected through a pipeline register `EX/RES`.

The `Control` block is responsible for setting control signals based on incoming commands, as well as receiver and transmitter status signals. Furthermore, the block is responsible for determining the opcode for the response message based on the incoming command as well as the current mode of operation (manual or autonomous). This approach to defining control signals is similar to that taken in [2, Ch.4]. As the name suggests, the `Response Compose` block is responsible for composing a response message from different sources based on the response opcode, as well as signalling the message transmitter to initiate transmission of a new message.

As can be observed from figure 4.5, the experiment setup's input signals have different origins. Most input signals are derived from the incoming command message, but the clock signal is driven by a `BUFGCE` clock gating buffer. This type of component is specific to Xilinx FPGAs. In its turn, the clock buffer's `Clock enable` signal is controlled by the `Setup Cycle Control` component. All sequential logic in the architecture is positive-edge-triggered, except for the logic driving the `Clock enable` signal. This approach of negative-edge-triggered logic was taken to ensure predictable timing for the clock edges of the gated clock signal. Figure 4.6 displays a timing diagram for the `Cycle Control` component working in combination with the `BUFGCE` clock buffer. By enabling the `Clock enable` signal while the clock signal is low, no unexpected edges will occur. The same applies to the disabling of the `Clock enable` signal.

Figure 4.6: Timing diagram for the `Cycle Control` logic block in combination with the `BUFGCE` clock buffer.



As described in section 3.4, the experiment setup is defined to take one clock cycle to perform read and write operations. In order to allow for this delay, a pipeline register was introduced to temporarily store the values passed from the `execute` stage to the `response` stage, as well as to handle the single cycle delay introduced by the negative-edge-triggered approach to gating the `experiment setup clock` signal.

4.4 Experimentation Package Specification

A format for the model's experimentation packages has been defined. This specification is similar to the experiment setup package specification as described in section 4.1. The package's unit of containment is also a `.zip` file archive and has a file structure identical to that displayed in figure 4.1. The experimentation package's manifest contents are mostly similar to that of the experiment setup pacakge's. It contains experiment setup metadata, as well as a description of the experiment setup's address space. Since the experimentation package is targeted at a specific FPGA development board, its manifest contains a description of this development board as well. In stead of referring to the source vhdl files, the manifest contains a file pointer to the bitstream file contained within the package.

4.5 Digilent Nexys 4 Composer

In order to facilitate the model's component composition process as described in section 3.5.1, a tool has been developed that automates this task. Similar to the experiment setup wrapper as described in section 4.2, this tool is implemented using the Java 8 programming language and its source code is made available through a git repository[9]. The tool is activated through the command line by the command `fpgaedu compose`.

The tool takes an experiment setup package as an input and extracts the experiment setup package's source files. The package's top-level file is parsed to extract its entity and architecture identifiers. This information is then used in the generation of a new top-level vhdl file that composes the experiment setup's sources and the controller sources to form a complete set of board-specific sources. A contraints file is then generated to match FPGA pins to the new top-level file's input and output signals.

In order to activate the FGPA's toolchain, a new script is then generated for the Xilinx Vivado SDK. This script instructs Vivado which files to include in the set of sources, as well as wihch specific FPGA model is to be targeted. Once generated, the script is executed in the background and a new bitstream is generated by Vivado. On successful execution, the bitstream is then packaged into an experiment setup package, together with a definition of the experiment setup's address space.

[9] `https://github.com/matthijsbos/fpgaedu-java`

CHAPTER 5

Evaluation

In order to evaluate the effects of the proposed model, it has been applied in different situation in order to evaluate its performance and behaviour. Simple example experiment setups have been implemented and tested, in order to analyze specific parts of the model's definition. For interaction with the controller command interface, a simple shell has been developed for the purpose of cycle control and manual address space inspection. The example experiment setup implementations, as well as the shell are available through a git repository[1].

5.1 Experimentation

For the purpose of testing the model's compatibility with experiment setups defined through combinational logic, an existing implementation of a 4-bit ALU was taken from the Internet. The current experiment setup wrapper implementation required some minor modifications to the ALU implementation's source code, due to the practical contraints as described in section 4.2. The source files were then processed by the experiment setup wrapper implementation in order to project its signals on an address space. After successful inspection of the process' output, the resulting files could then be processed by through the implementation of the component composition process in order to successfully create an experimentation package.

For functional validation of the experimentation package's contents, the bitstream file was manually programmed onto the Digilent Nexys 4 board's FPGA. Programming was done using Xilinx Vivado's hardware management tools. A connection between the PC and the controller's command interface was then established throught the shell program. Simple reading and writing commands allowed for interaction with the experiment setup's address space. The experimentation package's manifest file was used as a reference for the signal projections. Through manual modification of the experiment setup input signals' corresponding address values, it has proved possible to interact with the experiment setup logic. Through observation of address values corresponding the experiment setup's output signals, the ALU's correct functional behaviour could be validated.

In order to test the behaviour of experiment setup logic defined through combinational logic, an existing implementation of a shift register was taken from the Internet. Through the same process as described previously, the experiment setup's logic was processed into an experimentation package. After having programmed the FPGA, the shift register's functional behaviour could be validated. Through the manual operation the controller's cycle operation, one could accurately observe the shift register state transitions through the address value corresponding to its output port.

A final experiment has been implemented in order to test the effect of an experiment setup whose signals are projected onto multiple addresses, due to the signal's width being larger than that of a single address. A 26-bit counter was implemented and automatically processed through the experiment setup wrapping process. As could be observed in the generated experiment setup

[1] https://github.com/matthijsbos/fpgaedu-examples

wrapper logic, the counter's output signal was correctly projected on multiple addresses in the experiment setup's address space. After processing the file through the component composition process. Configuring the FPGA using the experimentation package's bitstream file showed the output signal to be correctly partitioned over multiple addresses as multiple cycles were executed.

5.2 Interaction

The concept of an address space as an means of interaction with the experiment setup's signal levels has proven to be a suitable abstraction. The functional behaviour of the tested experiment setups has been unaffected by this additional functionality and the concept of an address space has proven to be a reliable means of interaction. One could control different experiment setups without requiring modification of the shell used for experiment setup, since the controller's interface remained unchanged across different experimentation packages.

The utilization of FPGA clock buffers has shown to be an effective method for the control of state transitions in isolated blocks of logic. One could accurately observe the effects of the cycle operations in the experiment setup's address space and no negative effects were observed.

5.3 Abstraction Through Automation

The automation of the experiment setup wrapping process, as well as the component composition process has proven to be an effective method in reducing the time and work required in embedding experiment setup logic into a FPGA. In the development of the previous experiments, it has proven possible to create a properly functioning experimentation package within 15 minutes. This was achieved through the use of existing experiment setup implementations. Most of the time during the development of these packages was spent on modification of the implementation's source code, as the current implementation of the experiment setup wrapper is constrained in its support for VHDL language features. The automated processes generally complete within several minutes.

The perceived level of complexity as experienced during the development process has also significantly been reduced as a result of the automation these development processes, since a majority of the development process' operations have been reduced to two abstract operations. Although untested, the model's modifications seem to allow for a significant reduction in the knowledge and skills required in for operation of the development process.

CHAPTER 6

Conclusion

6.1 Conclusion

Through the development of a model, this thesis has presented a solution to the problems experienced in the application of FPGA development boards as a tool for experimentation. This thesis' model for the development of new experiment setups has shown to allow for reuse of components through the adoption of the concept of an address space. This concept is effective in the removal of the tight coupling between experiment logic and experiment controlling logic that can be identified in current work. This thesis' model of development has shown to allow for the automation of parts of the development process, hiding a significant part of this process' complexity. The resulting level of abstraction contributes to a separation of concerns as well as a reduction in the time required for development. This thesis' model for experiment setup interaction is based on the same concept of an address space, providing experimenters with a a method for control over their experiment setup's logic through a PC connection. This model of interaction hides the complexities of a FPGA and removes the need for a skills related to FPGA development.

6.2 Future Work

Different aspects of the model as proposed by this thesis have been explored and evaluated. Due to time limitations however, other aspects of the model have been left unexplored. The following sections will further elaborate on these areas, as they suggest possible points of entry for future work.

6.2.1 Board I/O Capabilities

Even though a FPGA development board generally contains a number of I/O devices for human interaction, these have not been included in the current model as described in chapter 3. The inclusion of these devices however, could further contribute to the development of an experimenter's physical view of the experiment setup. The omission of these devices in the current model does not mean that their inclusion has not been considered. Due to time limitations, their functionality has not included in the current model definition.

The type and number of available I/O devices is a specific property that is different for every type of development board. Since board-specific features have been considered to be handled by the controller's logic, the controller would have to be extended to support the input and output signals provided by I/O devices. The functionality assigned to these I/O devices may differ. One possibility would be to assign functionality that concerns the operation of the controller itself, such as cycle control.

Another possibility would be to assign functionality that allows for direct interaction with the experiment setup's address space. Since there exists no direct relation between the controller and the experiment setup, the mapping between the I/O device's signals and the experiment setup's

signal would have to be dynamic. For example, one would be able to dynamically set a specific led to correspond to a specific bit in the experiment setup's address space. Simple I/O devices such as leds and switches would easily correspond to a value in the experiment setup's address space. In the case of timing-sensitive I/O devices however, additional logic would need to be implemented in order for these device to function properly, since the experiment setup does not run on a regular clock. For example, one application of a LCD display would be to display the ASCII respresentation of an address range set to correspond to a specific range in the experiment setup's address space.

6.2.2 Experimentation Software

Although this thesis has developed a model for experimentation, the experimentation tools used for interaction are only suited for the experiments in this thesis. In order for the proposed model of interaction to be applied to a technically inexperienced audience, the development of a different type of user interface would be necessary. The address space description as included in an experimentation package is machine-readable and potentially allows for an automatic interpretation of the experiment setup's address space. As such, the address space can be presented to the user in a more user-friendly way.

The model currently does only present experiment setup signals as one or more bits and does not offer any information on how these values are to be interpreted. Signals might encode characters, integers or dates, for example. In order to allow for automatic interpretation of signals, the model would have to be extended to allow for this information to be added to an address space description.

6.2.3 Platform Independence

The achievement of platform independence has not been considered a goal during the development of this thesis' model and has not been tested as such. The model's definition does however offer handles in achieving this goal, since its definition has largely been established through generalization. In the context of this thesis, platform independence would mean that the same interfaces, tools and processes would apply to other development board and their related tools.

In order to apply the current work to a different board containing a FPGA from a different manufacturer, one would first need to adapt the controller implementation to be compatible with the board's communication device as well as the board's clocking resources for clock gating. Allowing for the automated execution of the component composition process, this thesis' implementation would need to be adapted to include these new sources, as well as the other board's SDK. True independence could potentially be achieved through the engineering of a plug-in architecture that allows for the isolation of any board-specific feature or setting isolated into a single unit.

6.2.4 Advanced Projection

Experiment setup input and output signals are not the only candidates for projection on the experiment setup's address space. The model of an address space is also compatible with the representation of an experiment setup's internal state elements, such as registers and memories. Achieving this sort of projection however, does not allow for the distinct separation between experiment setup logic and experiment setup wrapper logic, since it requires an integral approach to the development of an experiment setup component.

Although one might be required to integrate experiment setup logic and experiment setup wrapping logic, the resulting work could be packaged into an experiment setup package for automated processing in the component composition process, still allowing for the reuse of board-specific logic. In describing the experiment setup's address space, one might reserve a range of addresses to match the memory element's projection on the address space.

6.2.5 Advanced Automated Wrapping

The current implementation for the automated execution of the experiment setup wrapping process only considers the experiment setup logic's top-level input and output signals in the generation of an address space projection and corresponding logic. One improvement over the current implementation would be the automatic inclusion of internal signals in the generation of an address space projection. This would however require the automated modification of the experiment setup logic definition. Since signals are an explicit part of the VHDL language specification, this could potentially be achieved through automated syntax analysis and routing the signal to the experiment setup logic's top-level entity definition.

Another improvement would be the automated projection of memory elements such as registers and memories on the experiment setup's address space, so that their values could be inspected and modified in-between cycles. The recognition of memory elements in a specific HDL implementation is not a trivial task, since it relies on the recognition of specific language constructs. For a solution one could look at tools for FPGA logic synthesis. These tools apply a similar concept in the mapping of memory elements within the user's logic definition onto block rams, for example.

6.2.6 Application in Other Fields

The model described in this thesis is specifically developed to facilitate the application of FPGA development boards as a tool for experimentation in education. However, some of its features might be applicable to other fields as well. The means of interaction between a PC and logic contained within a FPGA through the concept of an address space might be of particular interest to other parties, since board-to-pc communication is a feature implemented in many projects.

Bibliography

[1] A. f. C. M. A. Joint Task Force on Computing Curricula and I. C. Society, *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. New York, NY, USA: ACM, 2013, 999133, ISBN: 978-1-4503-2309-3.

[2] D. A. Patterson and J. L. Hennessy, *Computer organization and design: The hardware/-software interface*. Newnes, 2013.

[3] R. Paharsingh and J. Skobla, "A novel approach to teaching microprocessor design using fpga and hierarchical structure," in *Microelectronic Systems Education, 2009. MSE'09. IEEE International Conference on*, IEEE, 2009, pp. 111–114.

[4] M. D. L. A. Cifredo-Chacon, A. Quiros-Olozabal, and J. M. Guerrero-Rodriguez, "Computer architecture and fpgas: A learning-by-doing methodology for digital-native students," *Computer Applications in Engineering Education*, n/a–n/a, 2015, ISSN: 1099-0542. DOI: 10.1002/cae.21617. [Online]. Available: http://dx.doi.org/10.1002/cae.21617.

[5] H. Oztekin, F. Temurtas, and A. Gulbag, "Bzk. sau. fpga10. 0: Microprocessor architecture design on reconfigurable hardware as an educational tool," in *Computers & Informatics (ISCI), 2011 IEEE Symposium on*, IEEE, 2011, pp. 385–389.

[6] A. El-Din and H. Krad, "Teaching computer architecture and organization using simulation and fpgas," *International Journal of Information and Education Technology*, vol. 1, no. 3, pp. 190–194, 2011.

[7] D. Jansen and B. Dusch, "Every student makes his own microprocessor," in *Microelectronics Education (EWME), 10th European Workshop on*, IEEE, 2014, pp. 97–101.

[8] M. Pereira, P. Viera, A. Raabe, and C. Zeferino, "A basic processor for teaching digital circuits and systems design with fpga," in *Programmable Logic (SPL), 2012 VIII Southern Conference on*, Mar. 2012, pp. 1–6. DOI: 10.1109/SPL.2012.6211804.

[9] C. M. Kellett, "A project-based learning approach to programmable logic design and computer architecture.," *IEEE transactions on education*, vol. 55, no. 3, pp. 378–383, 2012.

[10] J. H. Lee, S. E. Lee, H. C. Yu, and T. Suh, "Pipelined cpu design with fpga in teaching computer architecture," *Education, IEEE Transactions on*, vol. 55, no. 3, pp. 341–348, 2012.

[11] P. Bulić, V. Guštin, D. Šonc, and A. Štrancar, "An fpga-based integrated environment for computer architecture," *Computer Applications in Engineering Education*, vol. 21, no. 1, pp. 26–35, 2013.

[12] K. Nakano and Y. Ito, "Processor, assembler, and compiler design education using an fpga," in *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on*, IEEE, 2008, pp. 723–728.

[13] K. M. Al-Aubidy, "Teaching computer organization and architecture using simulation and fpga applications," *J. Comput. Sci*, vol. 3, no. 8, pp. 624–632, 2007.

[14] M. Holland, J. Harris, and S. Hauck, "Harnessing fpgas for computer architecture education," in *Microelectronic Systems Education, 2003. Proceedings. 2003 IEEE International Conference on*, IEEE, 2003, pp. 12–13.

[15] R. Owen. (Apr. 2015). MIPSfpga programme opens up the MIPS architecture to universities worldwide, Imagination Technologies Limited, [Online]. Available: http://blog.imgtec.com/mips-processors/mipsfpga-opens-up-the-mips-architecture-to-universities-worldwide.

[16] P. Kallstrom. (May 2010). Fpga cell example. File:FPGA cell example.png, [Online]. Available: https://commons.wikimedia.org/wiki/File:FPGA_cell_example.png (visited on 07/24/2016).

[17] *7 Series FPGAs Configurable Logic Blocks*, v1.7, UG474, Xilinx Inc., Nov. 2014. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[18] S. Gbor. (). About fpgas, [Online]. Available: http://home.mit.bme.hu/~szedo/FPGA/fpgahw.htm (visited on 07/25/2016).

[19] *Cyclone V Device Handbook, Volume 1: Device Interfaces and Integration*, CV-5V2, Altera Corporation, Jun. 2016. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cv_5v2.pdf.

[20] *7 Series FPGAs Memory Resources*, v1.11, UG473, Xilinx Inc., Nov. 2014. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

[21] A. Tanenbaum, *Computer networks*. Upper Saddle River, N.J: Prentice Hall PTR, 1996, ISBN: 978-0133942484.

[22] *Nexys 4 FPGA Board Reference Manual*, 502-274, Digilent Inc., Sep. 2013. [Online]. Available: http://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPNexys4/documenatation/Nexys4_RM_VB1_Final_3.pdf.

[23] E. Bendersky. (Aug. 2009). Framing in serial communications, [Online]. Available: http://eli.thegreenplace.net/2009/08/12/framing-in-serial-communications/ (visited on 05/10/2016).

[24] *7 Series FPGAs Clocking Resources*, v1.11.2, UG472, Xilinx Inc., Jun. 2015. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf.