

**The DEVBOX Development Education Platform:
An Environment for Introducing Verilog to Young Students**

by

Keith Lee

B.A.Sc, The University of British Columbia, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

January 2016

© Keith Lee, 2016

Abstract

Hardware description languages are considered to be challenging to learn, as are the logic design concepts required to effectively use them. University logic design courses are considered by many to be much more difficult than their software development counterparts and the subject is not generally addressed by pre-university curricula. Introducing hardware description earlier in a student's career will improve their chances of success in future logic design courses. The barrier-to-entry faced in introducing hardware description, caused by complex development environments and the learning of fundamental logic design concepts, must be overcome in order to facilitate a self-directed and interactive educational environment for young students. DEVBOX, an embedded System-on-Chip programming education platform, simplifies the learning process with its bare-bones development tools and interactive instructional material. It has been designed to be a self-contained, installation-free educational device with a browser-based interface that is self explanatory and easy to use. By including Verilog, a popular hardware description language, alongside the software programming languages in its repertoire, DEVBOX caters to the needs of introductory and intermediate logic design students in a dynamic self-directed learning environment.

Preface

This dissertation is original, unpublished, independent work by the author, K. Lee

Table of Contents

Abstract	ii
Preface	iii
Table of Contents	iv
List of Tables	viii
List of Figures	ix
Glossary	xi
Acknowledgments	xiv
1 Introduction	1
1.1 Motivation	1
1.2 Objective	4
1.3 Summary	4
2 Related Work	5
2.1 Programing Education Software	5
2.2 Programing Education Hardware	8
2.3 Teaching Strategies	12
2.4 Introductory Logic Design Education	15
2.5 Summary	15

3	The DEVBOX Platform Concept	17
3.1	Goal	17
3.2	Target Audience	18
3.3	Concept	19
3.3.1	User Experience	19
3.4	Device Hardware	27
3.5	User Interface	27
3.6	Device Software	28
3.7	Third-Party Development	30
4	The DEVBOX Prototype	35
4.1	Device	35
4.2	FPGA Hardware and I/O	37
4.3	Operating System and Drivers	41
4.4	Web Server and User Interface	45
4.4.1	The Back-End Code	46
4.4.2	The Front-End Code	48
4.4.3	TutoriML	50
4.5	Application I/O Interface	52
4.6	Compilers and Tools	54
4.6.1	C and GCC	54
4.6.2	Arduino C Runtime	55
4.6.3	Verilog Emulation and Verilator	58
4.7	Summary	65
5	Testing Procedures	67
5.1	Verilog Emulation	67
5.1.1	Compilation	68
5.1.2	Execution	70

5.1.3	Interpretation	72
5.2	Verilog Scalability	72
5.2.1	Compilation	73
5.2.2	Execution	73
5.2.3	Interpretation	76
5.3	Course Material	76
5.3.1	Results	77
5.3.2	Interpretation	82
6	Future Work	83
6.1	Web Interface	83
6.1.1	Editor and Debugging	83
6.1.2	Tutorials	84
6.1.3	Templates	84
6.2	Verilog Simulation	85
6.2.1	ZUMA eFPGA	85
6.2.2	Debugging	85
6.3	Community Websites	85
6.4	Device Prototype	86
7	Conclusion	87
	Bibliography	90
A	Arduino Runtime Source	93
B	Node.js Server Source	101
C	Tutorials.php Source	104
D	Logic Design VHDL Assignments	111
D.1	2013 Cohort	111

D.2 2014 Cohort 121

List of Tables

Table 3.1	Initial Discovery Steps for New Software/Hardware Development Students	18
Table 3.1	Tutorial Use Case	22
Table 3.2	Project Use Case	23
Table 3.3	Help Resources Use Case	25
Table 3.2	Requirement-to-Component Mapping	27
Table 3.3	Langtable Example	31
Table 4.1	ARM Cortex A9 Technical Specifications	37
Table 4.2	DEVBOX Physical Address Map	44
Table 4.3	TutoriML Tags	50
Table 5.1	Control PC Specifications	68
Table 5.2	Test Verilog Design Complexity	69
Table 5.3	Lab 1 Assignment Results	78
Table 5.4	Lab 2 (2013) Assignment Results	79

List of Figures

Figure 1.1	Minecraft’s Redstone Logic Gates	3
Figure 2.1	The SCRATCH Development Environment	6
Figure 2.2	Alice Development Environment	7
Figure 2.3	Codeboard.io Integrated Development Environment (IDE)	8
Figure 2.4	One Laptop Per Child (OLPC) XO Laptop (src: wikipedia.org)	9
Figure 2.5	Raspberry Pi Models	10
Figure 2.6	BeagleBone Black	10
Figure 2.7	Arduino Uno Board	11
Figure 2.8	Kano Kit	12
Figure 2.9	Embedded Software Development Device	13
Figure 2.10	AEIOU Interface	14
Figure 2.11	1:1 Mapping of Agile Concepts to Teaching Methods	14
Figure 3.1	XBOX One: a Modern Gaming Console (src: www.xbox.com)	19
Figure 3.2	DEVBOX Design Concept Overview	20
Figure 3.3	Form-Factor Analogs	32
Figure 3.4	User Interface Elements	33
Figure 3.5	DEVBOX Software Component Hierarchy	33
Figure 3.6	Standardized Tutorial File Structure	34
Figure 4.1	The Terasic DE-1 SoC Board (image: www.terasic.com.tw)	36

Figure 4.2	DE-1 SoC Layout (image: www.terasic.com.tw)	38
Figure 4.3	DEVBOX Prototype Workstation	39
Figure 4.4	DEVBOX Prototype FPGA System Diagram	40
Figure 4.5	VGA Controller IPs	42
Figure 4.6	User Space Versus Kernel Space	43
Figure 4.7	Character Device Driver Software Structure	44
Figure 4.8	The DEVBOX Tutorials Page	47
Figure 4.9	DEVBOX Prototype Index Page	49
Figure 4.10	The Arduino Visualization Screen	57
Figure 4.11	Verilog Port Assignment Web Form	59
Figure 4.12	AvalonMM Master Block Diagram	62
Figure 4.13	Avalon MM Slave Block Diagram	62
Figure 4.14	Avalon MM read/write Waveform	63
Figure 4.15	Emulating Avalon-MM Master	64
Figure 5.1	Verilator Build Times	70
Figure 5.2	Verilator Execution Times	71
Figure 5.3	Scalability of Compilation for ARM Verilator	74
Figure 5.4	Scalability of Execution for ARM Verilator	75
Figure 5.5	Real vs. CPU Execution Times	75
Figure 5.6	VGA Adapter Core as a Black Box	80

Glossary

AE-IS Agent-based E-learning System	ECE Electrical and Computer Engineering
AEIOU Ambiente para la Ensenanza Integral de Objetos en Universidades	eFPGA embedded FPGA
AJAX Asynchronous JavaScript and XML	FAQ Frequently Asked Questions
API Application Program Interface	FF flip-flop
AXI Advanced eXtensible Interface	FPGA Field Programmable Gate Array
CPLD Complex Programmable Logic Device	FSM finite state machine
CPS Cycles per Second	GB Gigabyte
CPU Central Processing Unit	GCC The GNU C Compiler
CSS Cascading Style Sheet	GPGPU General-Purpose Graphics Processing Unit
DAC Digital-to-Analog Converter	GPIO General Purpose I/O
DDR3 third-generation Double Data Rate	GUI Graphical User Interface
DE Development and Education	HAL Hardware Abstraction Layer
DMA Direct Memory Access	HD High Definition
DSP digital signal processor	HDL Hardware Description Language

HDMI High-Definition Multimedia Interface	MMU Memory Management Unit
HPS Hard Processing System	OLPC One Laptop Per Child
HTML HyperText Markup Language	OOD Object Oriented Design
HTTP HyperText Transfer Protocol	OOP Object-Oriented Programming
Hz Hertz	OS Operating System
I/O Input/Output	PC Personal Computer
IC integrated circuit	PHP PHP: Hypertext Preprocessor
IDE Integrated Development Environment	RAM Random Access Memory
IP Intellectual Property	RGB Red-Green-Blue
ISA Instruction Set Architecture	RGBA RGB-Alpha
JS JavaScript	RTE Run-Time Environment
KB Kilobyte	RTL Register-Transfer Level
kHz kiloHertz	SD Secure Digital
LCD Liquid Crystal Display	SDRAM Synchronous Dynamic RAM
LE logic element	SoC System on Chip
LED Light Emitting Diode	sof SRAM Object File
LKM Linux Kernel Module	TCP/IP Transmission Control Protocol/Internet Protocol
LUT look-up table	TutoriML Tutorial Markdown Layer
MHz Megahertz	UART Universal Asynchronous Receiver/Transmitter
MIME Multipurpose Internet Mail Extension	USB Universal Serial Bus
MM Memory Mapped	

UX User eXperience

VTR Verilog to RTL

VGA Video Graphics Adaptor

WPS WiFi-Protected Setup

VPL visual programming language

Acknowledgments

My gratitude to UBC, the Faculty of Applied Science, and the Electrical and Computer Engineering department for the opportunity to grow and learn, to Dr. Guy Lemieux for believing in me and for his patience, and to Dr. Lutz Lampe and the Faculty of Graduate and Postgraduate Studies for opening the door.

To my parents, Derek and Nancy, I thank you for your unwavering support and understanding as I navigated the most challenging time of my life. I owe you two a great deal.

Most importantly, to my wife, Brandi: you have been my guiding light these past six years. I would not have made it this far without you.

Dedicated to my children, Payton, Garrett and Isla.

May all doors in life be open to you.

Chapter 1

Introduction

Hardware Description Languages (HDLs) are text-based methods of describing the behavior of digital logic circuits, much like a software programming language describes the behavior of an application. The internal design of a Central Processing Unit (CPU) or a Liquid Crystal Display (LCD) controller are two examples of hardware behaviours easily described by HDLs.

With the right tools, HDLs can be used to specify a digital implementation for Field Programmable Gate Arrays (FPGAs), which are integrated circuits (ICs) designed to implement logic circuits. They can be programmed and reprogrammed to the user's specifications. This is accomplished by using a large array of logic elements (LEs), each containing one or more look-up tables (LUTs) and flip-flops (FFs), math-oriented digital signal processors (DSPs) and memory blocks on-chip. These elements can be configured to interact, producing a desired logic behaviour. HDLs, as a part of logic design, are required learning in the field of Electrical and Computer Engineering (ECE), and are of increasing interest in computer science coursework. They are, however, considered to be challenging to learn.

1.1 Motivation

Many of the challenges faced by introductory logic design students are analogous to those faced by new computer programmers. They must acquire the necessary software tools to generate compiled output and learn to use them. While software programming with Integrated Development Environments (IDEs) can be complex, the development environments for HDLs and the devices they program

are even more so.

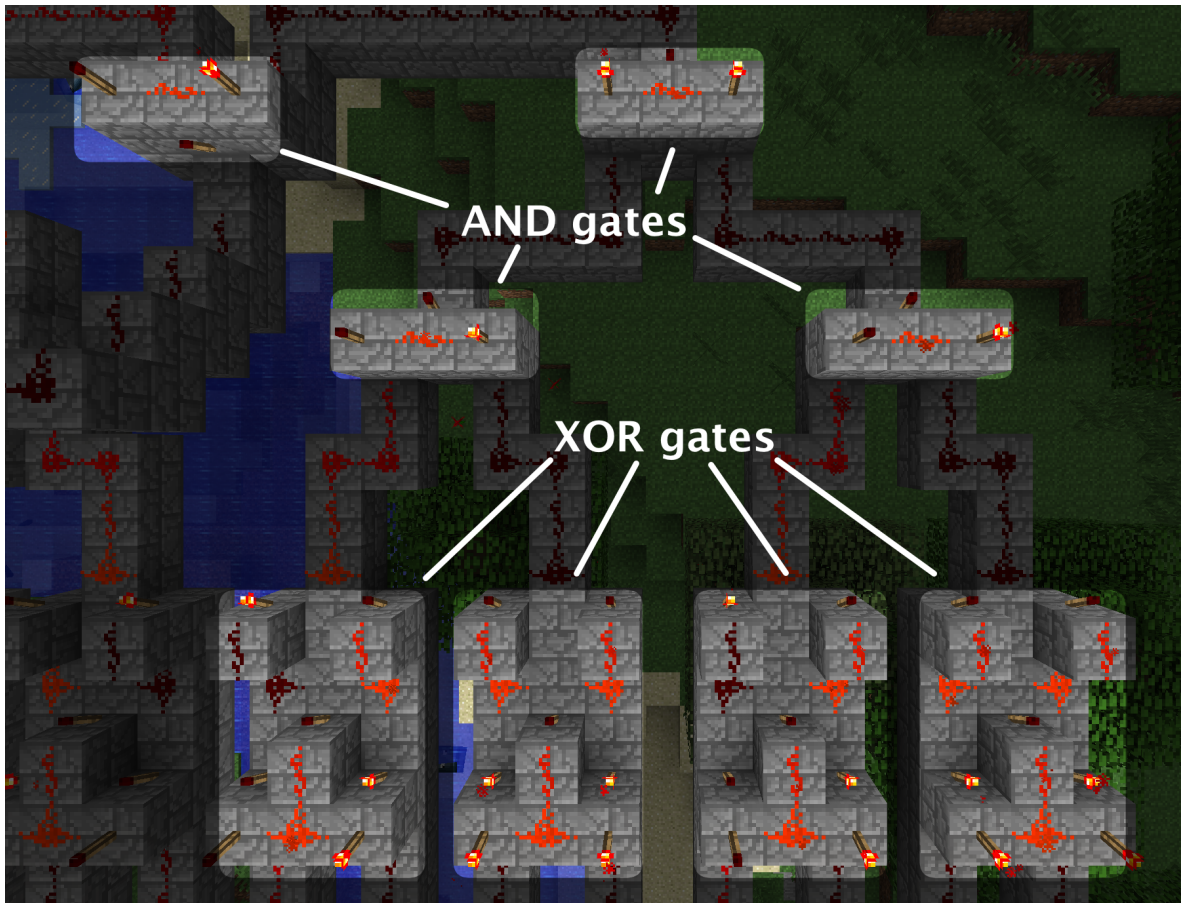
Once students have acquired the tools and the knowledge to use HDL development tools, they must find resources to guide them through the learning process. Often, these resources come in the form of text-based or video tutorials. Videos distract from the learning environment and often move at an accelerated pace, forcing viewers to pause the video frequently and watch important sections repeatedly. Text-based tutorials allow readers to follow at their own pace, but still remove them from the tools they are using.

Computer programming courses have been offered at some high schools as electives for many years, introducing students early on to software development concepts. Languages using simplified syntax and interaction models like SCRATCH [29], a visual programming language (VPL) for children, have helped introduce computer programming to students as young as age eight. However, HDLs, despite requiring a similar learning process, are considered to be more difficult, and are generally not introduced until advanced university courses. Introducing logic design concepts and HDLs at an earlier age, such as high school, will result in a better understanding of contemporary computing challenges in parallelism and multiprocessing. As industry continues to have difficulty increasing CPU clock frequency above the 4 GHz mark, and hardware accelerators, such as General-Purpose Graphics Processing Units (GPGPUs) and FPGAs, become increasingly important in the foreseeable future, logic design skills will become a valuable asset to programmers.

Many high school students have been unintentionally teaching themselves the fundamentals of combinational logic. Minecraft [9], the popular sandbox-style exploration video game, incorporates elements known as redstone blocks as analogs to logic circuits. Players can build simple logic gates by connecting various redstone blocks, as seen in Figure 1.1. The creations assembled from these circuits can be as simple as a door-opening switch, or as complex as a calculator [2]. Since students are already gaining a rudimentary understanding of logic and microcircuitry, introducing HDL is a sensible and complimentary next step.

Further, in most university-level degree programs for computer science and ECE, development using Verilog and VHDL, two popular HDLs, is under-emphasized in comparison to software development. It is introduced late in the curriculum or not at all and presents many teaching strategy

Figure 1.1: Minecraft's Redstone Logic Gates



challenges. Hands-on learning is a valuable tool for both software and hardware design education and promotes “*learning by doing*” in group or lab settings as well as independently. The tools and resources for software development are free, numerous, and readily available on the worldwide web to the point of excess. Conversely, the tools necessary for hardware design are sparse, prohibitively expensive, device-specific, and are often produced and distributed by FPGA manufacturers themselves. For example, Altera and Xilinx, who maintain a combined FPGA market are greater than 85%, each have their own commercial products for developing and synthesizing hardware designs. These include custom versions of peripheral applications like Mentor Graphic’s ModelSim simulation software. Some additional simulation and verification tools exist, including Verilator, a free and open-source Verilog-to-C++ compiler and simulator, but neither these nor FPGA developers’ proprietary tools are designed with HDL education in mind; they are large and complex, can impose licensing

constraints and can have challenging configuration problems.

1.2 Objective

Despite the proven value of hands-on education techniques [17] [23] and the potential benefit of introducing Verilog or VHDL early on, very little has been done to facilitate self-directed education and experimentation for logic design. To this end, this thesis has created a single development and education environment designed to teach traditional programming languages, such as C/C++, as well as hardware design with Verilog at an introductory to intermediate level. The proposed platform, dubbed DEVBOX, incorporates all of the tools necessary for this broad range of tasks onto an ARM-based FPGA development board. The preconfigured DEVBOX platform is easy to use and works out of the box, thus simplifying the setup procedure and allowing prospective students to focus entirely on the learning process.

Specifically, this thesis sets out to:

- (a) create a portable and easy-to-use development education platform,**
- (b) demonstrate a dynamic IDE and training environment mock-up,**
- (c) implement Verilog emulation functionality, and**
- (d) evaluate the platform's viability for use as an HDL education tool.**

1.3 Summary

Although the barrier to entry for introductory computer programming is steep, the challenges faced in introducing HDLs are even greater. It is thus desirable to simplify the initial learning stages and target pre-university students as well as early-year university students. DEVBOX has been conceived to serve this purpose. This flexible and portable platform is capable of presenting a library of tutorials and a development environment for multiple computer languages, including Verilog, with virtually no setup time. With DEVBOX, students will be able to teach themselves the fundamentals of logic design without dealing with the complexity of current FPGA development platforms.

Chapter 2

Related Work

This chapter provides an overview of work related to computer programming education and logic design software, hardware and strategies. A common strategy employed within these works is that of simplification. The common goals among these resources are either to make the initial learning process as simple as possible or to make the acquisition of the necessary tools as easy as possible.

Many discussions on the best programming paradigm to which students should first be exposed are also present, but no common conclusion is prevalent. Some applications employ an imperative-first approach while others focus on object-first instruction models.

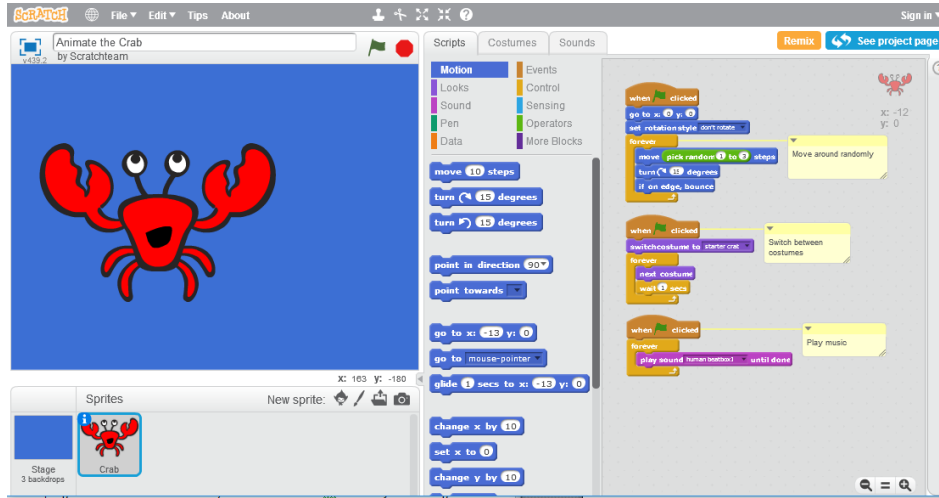
Imperative-first instruction focuses on the syntax, sequence and structure of commands in an action-driven runtime environment. Its strength lies in a more concrete and immediate feedback loop in a trial-and-error debugging process.

Object-first instruction, as the name suggests, focuses on Object-Oriented Programming (OOP) and Object Oriented Design (OOD). Teaching the concept of classes and objects, member functions, inheritance and polymorphism. This paradigm is prevalent in professional programming environments.

2.1 Programing Education Software

Alice [18] [19] and MIT's SCRATCH [32] [28] [29] are VPLs designed to instruct young students in the fundamentals of software programming. VPLs allow the student to interact with their code

Figure 2.1: The SCRATCH Development Environment

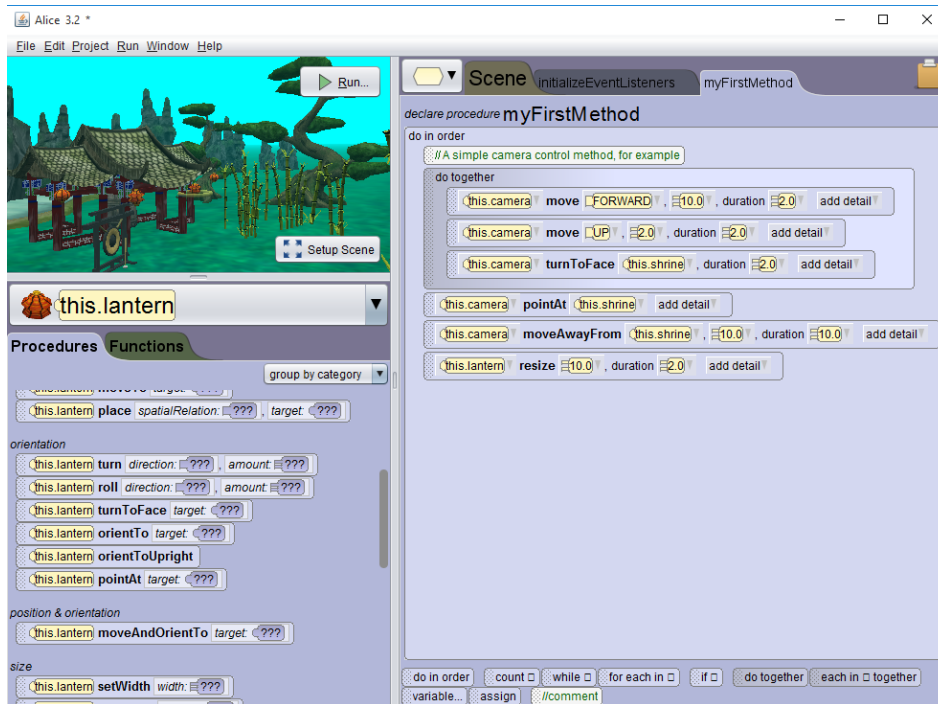


in a more tangible way by observing the interactions of various programming elements visually. In another example, Greenfoot [25] is not strictly a VPL, however it does present OOP in a very visual way. Online resources, such as Codeboard.io [6] are also available to connect educators with their students. These applications are described in greater detail below.

SCRATCH and its online community have steadily grown in popularity among grade-school instructors and students since its public release in 2007 and has been included in several programming start-up kits. It follows an imperative-first instruction model with a similar code structure to C. Tailored to the needs of students aged 8 to 16, its colourful command blocks, single-page interface, live feedback, tinkerability and lack of error messages make interacting with the development environment fun. Command blocks fit together like puzzle pieces, as seen in Figure 2.1, eliminating syntactical errors by permitting only valid sequences of commands in the script. They control the behaviours of “sprites” on a “stage”. Adjusting variables and parameters is integrated into the visual nature of the VPL, making use of drop-down boxes and text fields within the control modules.

Alice is a multi-paradigm visual programming language for 3D software development aimed at novice programmers. It includes many of the features of SCRATCH, including drag-and-drop command blocks and visual parameter editing, but is targeted to a much older audience. It introduces more advanced concepts, such as OOP, and strips away the puzzle-piece design that forced syntax and eliminated error messages in SCRATCH. It follows a UI design similar to SCRATCH providing

Figure 2.2: Alice Development Environment



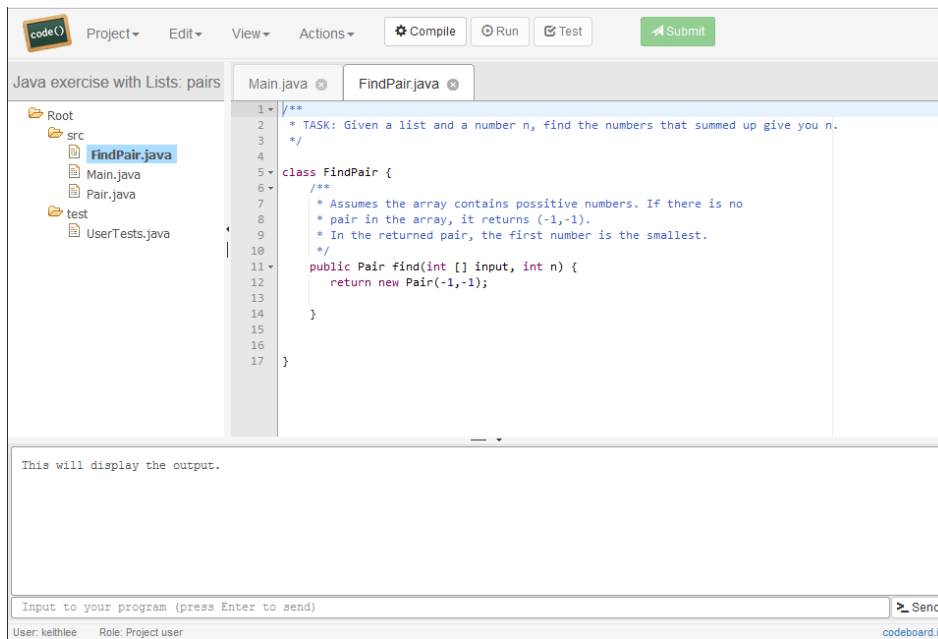
a single page interface and minimizing the frequency of pop-up windows. Figure 2.2 illustrates the Alice user interface (UI).

Greenfoot teaches object oriented design and programming by presenting a visual representation of the students' classes and their inheritance trees. Similar to SCRATCH, projects are divided into two super-classes. The "World" is a parent class for the execution area of a given project, or "scenario". The "Actor" superclass provides the behavioural methods for the instantiation, placement and movement of actors within the world. An extensive code highlighting scheme helps to emphasize and clarify the structure of a class and its member functions.

Codeboard.io [6] is an online IDE that allows educators to share programming exercises with students and automate testing and marking in an extensive array of languages. The site uses unit tests written by the educator to analyze the validity of a student's submission and generates a mark based on $\frac{\text{tests passed}}{\text{tests failed}}$. Figure 2.3 presents the primary in-browser development interface, including project navigator, console Input/Output (I/O), and a drop-down menu providing a limited selection editor options. The IDE design emulates a layout common to many development applications, such as

Eclipse [8] and Code::Blocks [5].

Figure 2.3: Codeboard.io IDE



2.2 Programming Education Hardware

Companies and organizations are continuously providing new computer hardware platforms for programming education, many utilizing the software elements described in the previous section. From cheap preconfigured laptop computers to microcontroller-based Maker boards, a large number of educational resources exist for young programming students.

Codestarter [31] is a non-profit organization whose primary goal is to provide young and underprivileged prospective programmers with the tools required to learn to code. By reconfiguring inexpensive Chromebooks with the required tools by a software developer and delivering them to under-privileged grade-school students with the help of community donations, the Codestarter team strives to “[...] put a developer-friendly laptop into the hands of every kid that wants to learn how to code”. By making the tools used to convert these Chromebooks open-source and readily available, anyone can easily obtain a programmer-ready PC for under \$200.

One Laptop Per Child (OLPC) [10] is another non-profit organization that strives to bring laptops to young students in developing countries. Figure 2.4 presents the XO laptop, a \$35 power-efficient,

Figure 2.4: OLPC XO Laptop (src: wikipedia.org)

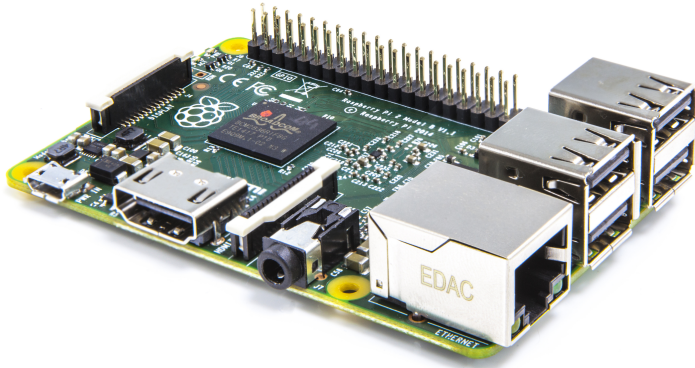


sturdy, and cost-effective computer. This is OLPC's flagship product, distributed world-wide to help children of impoverished communities learn technical skills. With alternative power sources, such as solar panels and hand cranks, the XO can be used and charged without access to a power grid.

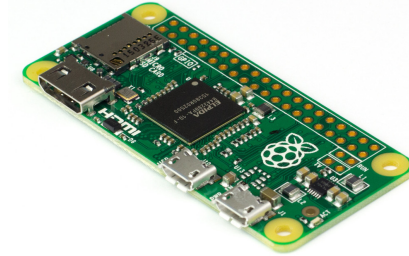
The Raspberry Pi [11] was developed by a collection of professors and researchers at the University of Cambridge concerned with the decline of **CS!** (**CS!**) applicants' programming experience. The concept is to encourage young people to experiment with computers. The Raspberry Pi 2 in Figure 2.5a is a recent device from The Raspberry Pi Foundation. It is a full-featured ARM-based computer for less than \$45. The Raspberry Pi Zero, their latest device pictured in Figure 2.5b, is a miniaturized version of their original Pi for only \$5.

Beagleboards [3] are uncannily similar to the Raspberry Pi in design and functionality. The primary differences are their purpose and I/O devices. BeagleBoards are designed for Makers, a class of hobbyists and tinkerers, to control homebrew electronic devices. It therefore has a wider array of GPIOs with which these devices communicate with hardware and circuitry. This makes the development of embedded applications more appealing and accessible to the general public. The BeagleBone

Figure 2.5: Raspberry Pi Models

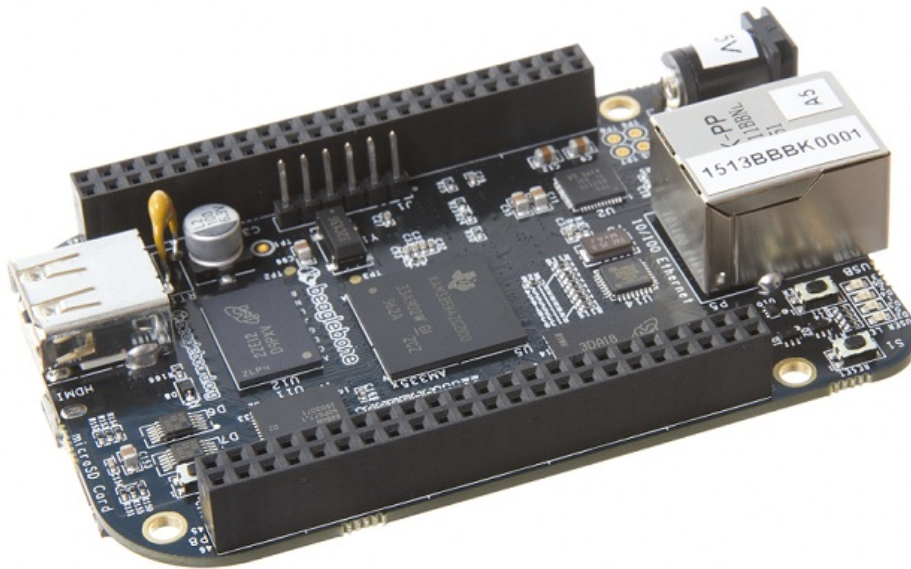


(a) Raspberry Pi 2



(b) Raspberry Pi Zero

Figure 2.6: BeagleBone Black

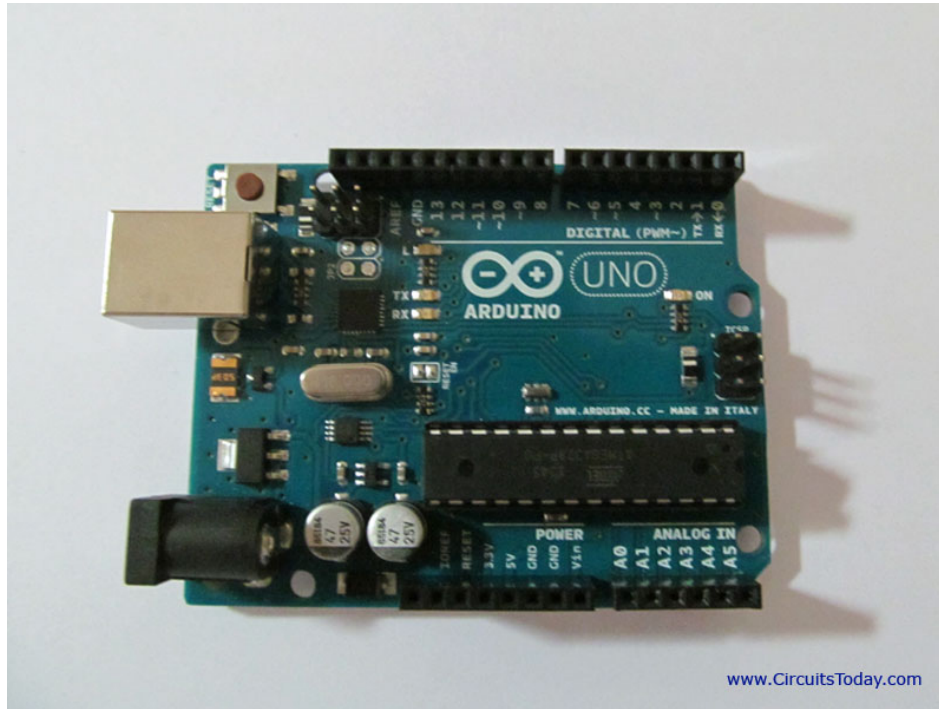


Black (Figure 2.6), starting at \$60, is their small-scale embedded platform.

Arduino [1], like the BeagleBone, is designed for Makers and artists to control custom electronic devices. Its simple Application Program Interface (API) and IDE, reduced complexity, and active development community make it the ideal platform for learning embedded programming. The Uno (Figure 2.7), Arduino's most successful device, is built around the ATmega328P microcontroller.

Kano [24] provides an inexpensive Raspberry Pi-based kit for young computer enthusiasts for

Figure 2.7: Arduino Uno Board



\$100. It gives them the opportunity to “*Build your own computer*” and customize it to their liking. Its flash memory comes pre-loaded with Linux and a variety of games, game-making tools (such as SCRATCH), and useful apps like the Geany IDE [33], media editors and a web browser. The goal with Kano, like Codestarter, is to make acquiring a developer-ready computer easy and inexpensive for kids. It has the added benefit of being fun to make, which adds to its appeal.

The specialization of computer hardware to the task of programming education is relatively unexplored in the academic community and there are not many published papers on the subject. In the English-translated abstract of their 2014 journal article [30], Tachi et al. describe teaching embedded software development using open source hardware, emphasizing the quick adoption of new technologies in embedded systems. The device, shown in Figure 2.9, appears to be developed specifically for the course and challenges students to analyze technical documents in order to develop embedded applications.

Figure 2.8: Kano Kit

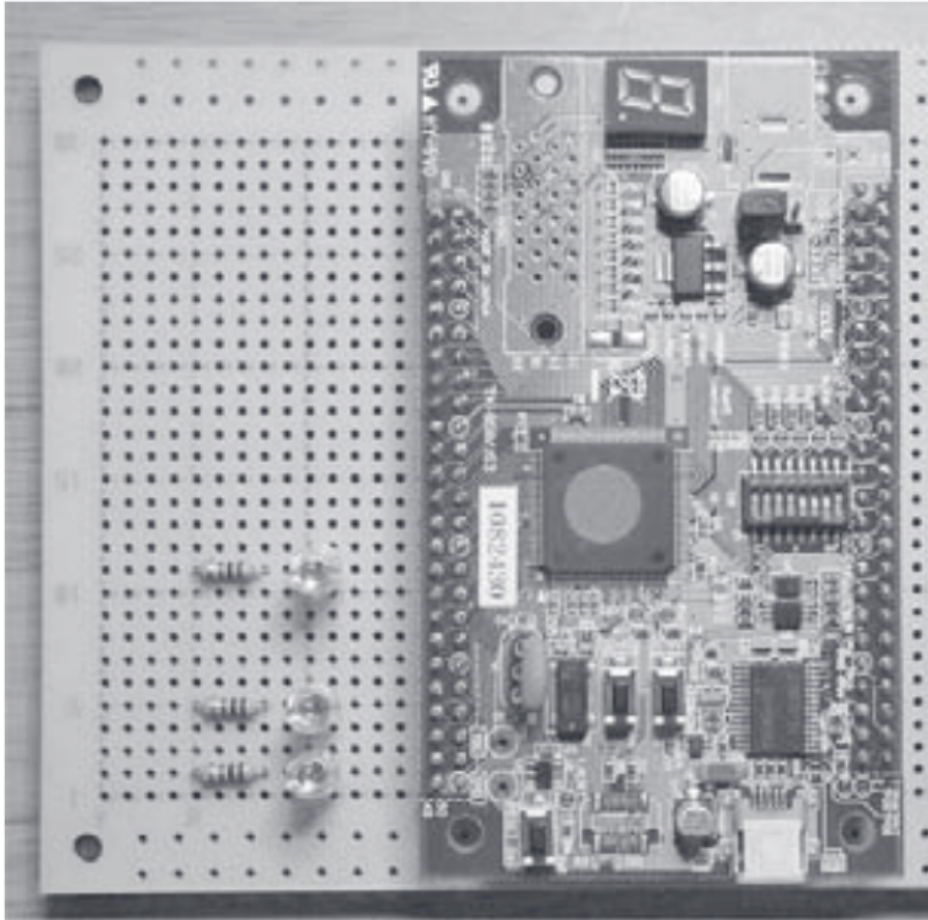


2.3 Teaching Strategies

Koulouri et al. [26] recently performed a study quantifying the effectiveness of various approaches to introductory computer programming by evaluating the importance of language selection, instructor feedback, and preparatory workshops in decreasing the fail/drop rate among first-year **CS!** students. The study iteratively modifies an introductory **CS!** curriculum, replacing Java with Python as the primary programming language. The work shows that the use of a syntactically simple programming language, such as Python, has a positive impact on the complexity of software created by students while also decreasing the dropout/fail rate within introductory **CS!** courses. It also motivates the use of individual formative feedback from instructors and illustrates the importance of preparatory education.

Licea et al. [27] have developed and deployed a localized Java education platform, Ambiente para la Enseñanza Integral de Objetos en Universidades (AEIOU), designed to gradually introduce OOP to Mexican engineering students. The AEIOU platform includes three tiers of complexity to accommodate novice, intermediate and advanced students and presents the interface, help material and compiler output in Spanish. This offers a much-needed relief of the language barrier that exists between non-English speaking students and the development of software. While the concept of a

Figure 2.9: Embedded Software Development Device



three-tier programming interface allows for users to ease into the task of object oriented programming and design, its localization limits its relevance to a relatively small population.

Bassey Isong [21] discusses the importance of reformulating introductory computer programming courses to align the challenges in teaching and learning computer programming to the Agile development process in a dynamic and lab-oriented curriculum. The paper aligns each of the elements of the Agile manifesto to a teaching methodology and combines them to create a more effective course configuration. A combination of pair-programming, lab-focused instruction, active instructor feedback, and immediate course improvements lends itself to a modern and student-centric introductory programming course.

Ljubomir Jerinic [22] suggests that learning from making interesting mistakes could be a power-

Figure 2.10: AEIOU Interface

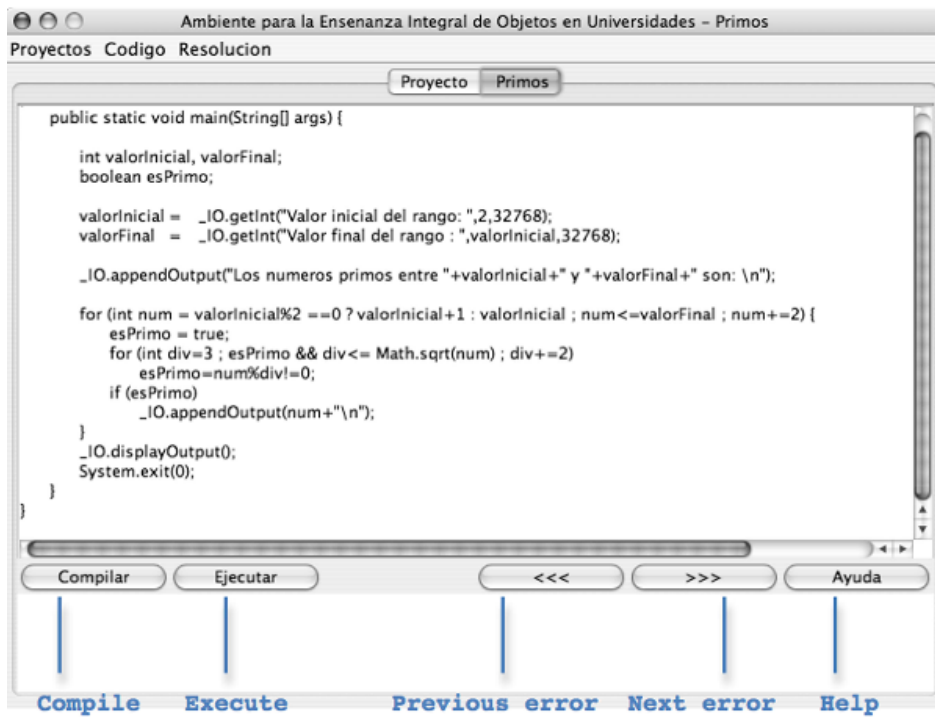
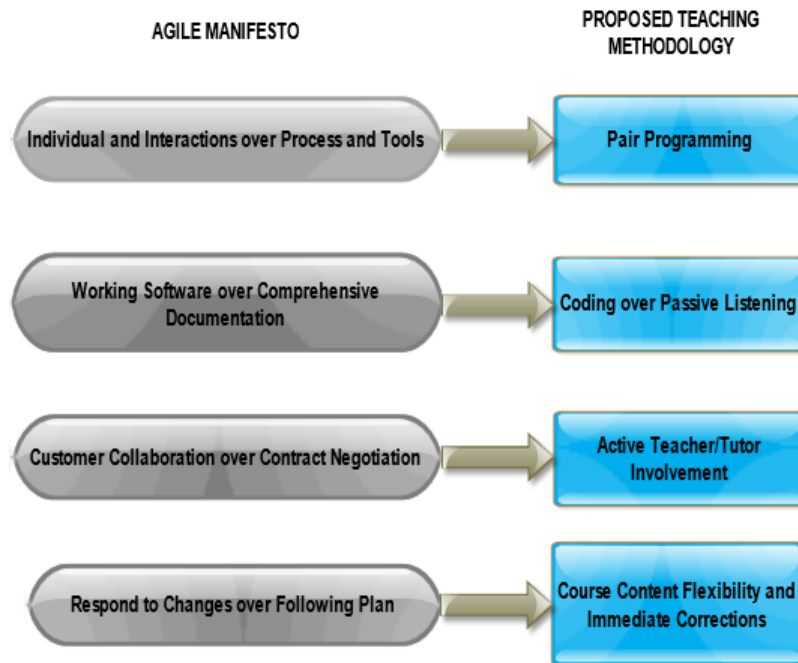


Figure 2.11: 1:1 Mapping of Agile Concepts to Teaching Methods



ful teaching tool. He proposes an Agent-based E-learning System (AE-IS) that employs two software agents — autonomous computer programs designed to assist users — to guide students through programming tasks. One agent provides helpful tips and suggestions as the student progresses through the assignment. The other is designed to mislead the student into making mistakes. The use of pedagogical patterns is also considered vital to assignment creation, capturing best practises in the given domain.

2.4 Introductory Logic Design Education

Cifredo-Chacn et al. [17] recommend a first-year course curriculum that teaches students computer architecture, fundamental VHDL, and FPGA development. By eliminating all device-specific and non-synthesis VHDL elements, the course material complexity is reduced to accommodate less experienced students.

Yen-Chu Hung [20] demonstrated that students given a problem-solving primer before learning Verilog scored better on final course examinations than those who received no additional training. For the first three weeks of classes, one test group was taught problem solving by deduction and another group problem solving by analogy. The control group was given lessons in word processing instead. On average the control group scored significantly lower on the experiment's post-test than the two experimental groups.

2.5 Summary

The academic community has invested a notable amount of time to identifying the primary challenges in early computer programming education. The result of these studies has been a variety of languages, such as SCRATCH and Alice, and comprehensive curricular mechanisms, including preparatory seminars and the use of syntactically simple programming languages, designed to address these road blocks.

The academic community has largely ignored the implementation of specialized hardware devices for programming education, leaving the field open to the tech industry. Organizations such as OLPC and Codestarter strive to supply programming-friendly laptops to students in need. Products like the Raspberry Pi are ideal for young developers and are included in kits designed specifically for this task.

All of these technologies and strategies are directed at introductory software development for young students, but do not touch on the subject of logic design, HDLs, or computer architecture. Materials discussing the early education of hardware developers is practically unexplored and limited to freshman and sophomore university students. Despite evidence that pre-university programming courses reduce attrition in university **CS!** and ECE programs, no effort has been made to test the same of hardware description.

Chapter 3

The DEVBOX Platform Concept

3.1 Goal

The primary goal of the DEVBOX platform is to provide a single platform that contains the resources necessary to begin learning a software programming or hardware design language with minimal effort by the student. This is meant to reduce the initial startup time and burden of decision in early stages, thereby reducing the barrier to entry for new developers. In order to present an improved alternative to the current model, the decision and setup process – from inception to fruition – must be as simple and quick as possible. Table 3.1 describes the typical setup process for new developers compared to the simplified procedure proposed by the DEVBOX model. The steps involved in the typical method are greater in both number and complexity.

To my knowledge, this approach to HDL and logic design education is unprecedented, as is the presentation of Verilog to pre-university students. An early introduction to logic design and HDL with a low-overhead, self-directed approach will give aspiring ECE and computer science students a solid foundation on which they can build their understanding of computer architecture, Systems on Chip (SoCs), and parallel C.

The simplicity of a standardized hardware with plug-in software is analogous in many ways to the model of gaming consoles (Figure 3.1). When an individual purchases a new title for his console, he knows that it will “just work,” as the software is developed to match the exact specifications of the console itself. Additionally, little more than inserting a cartridge or disc is required to install

Table 3.1: Initial Discovery Steps for New Software/Hardware Development Students

Typical	DEVBOX
<ul style="list-style-type: none"> • Select programming language • Choose, download/install IDE • Install missing tools and libraries and resolve software conflicts • Install device drivers and resolve OS conflicts • Search for and choose training resource online • Select tutorial • Read instructions • Copy example code from browser to IDE • Test and modify code • Install additional libraries as needed by tutorial content to resolve build errors 	<ul style="list-style-type: none"> • Connect device • Select programming language • Select tutorial • Begin reading instructions while testing and modifying example code in single window

the game and instructions for its use are provided by the device itself. Comparatively, DEVBOX setup requires the insertion of a micro SDcard, connection to a High-Definition Multimedia Interface (HDMI)-enabled display, and WiFi access.

3.2 Target Audience

The parallels between the DEVBOX model and that of gaming consoles does not end with functional similarities. It also overlaps the target age range for the platform, generally grade school students aged 13 to 18. Some students in this demographic have a passing to strong interest in computer programming or hardware, or intend to enter the field of computer science or software engineering after graduation. Their experience with basic computer skills ranges from novice to intermediate: enough to use a browser, but anything more complicated than Office software is usually foreign.

A secondary intended user is the educator. This includes teachers, tutors, and professors. The educator may be instructing grade school students in fundamental programming, or providing supplementary resources for more advanced students with regards to new programming models. They may want to add tutorial resources or include new languages to the platform.

Figure 3.1: XBOX One: a Modern Gaming Console (src: www.xbox.com)



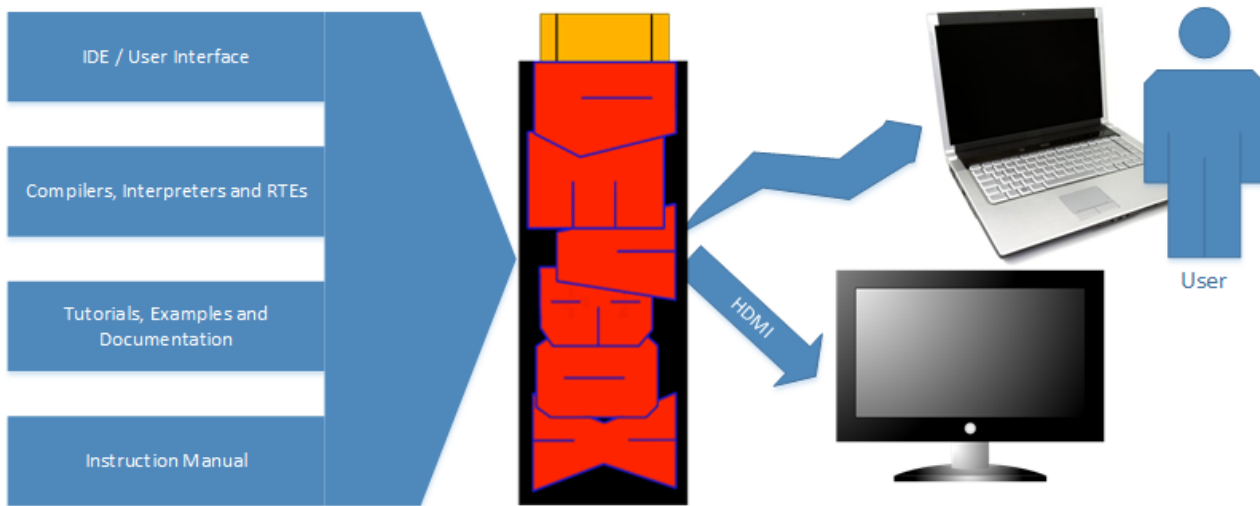
3.3 Concept

In order to meet the goal described above, DEVBOX must be portable, inexpensive, easy to use, and self-explanatory. To maximize portability, the DEVBOX device is designed to be a self-contained host for interaction and execution. It requires no access to the Internet and is agnostic to the user's personal computing device. Its only external dependencies are upon a local WiFi network and a HDMI connection to a display device. Running its own web server, its use comes from an intuitive web-based interface, developed to guide users through its pages as well as the development learning process. With a price tag below \$50, it would be widely accessible. Figure 3.2 provides an outline of the overall design concept.

3.3.1 User Experience

It is important to design DEVBOX user interaction to be as self-explanatory as possible in order to take into account the range of skill the target audience will have. A large number of users are likely to have

Figure 3.2: DEVBOX Design Concept Overview



minimal exposure to programming development software and training material and will require visual and textual cues in order to complete desired tasks. Use Cases 3.1 and 3.2 describe the interaction model for the two primary tasks to be performed by users. These procedure descriptions help identify platform features.

Use Case 3.1 assumes that the user is interacting with the device for the purpose of HDL training for the first time. This stems from the assumption that the user's initial task will be to locate and access training materials before developing HDL modules independently. The preconditions for this procedure would be described through some aspect of the device's packaging, such as a temporary decal, or a box insert. The initial setup procedure will be as follows:

1. Insert microSD card into device.
2. Connect device to HDMI port of display.
(this step may include connecting an external power source.)
3. Turn device on and link to WiFi via WiFi-Protected Setup (WPS).

It is meant to be no more complicated than setting up a video streaming device such as the Chromecast stick illustrated in Figure 3.3a. WPS is now a common feature among wireless routers that simplifies

the WiFi connection setup process. It allows for one-button passcode-free configuration of a device's connection settings.

In the process of accessing training materials, the student must interpret connection instructions displayed on the device's connected display. The described steps must be concise but complete. Visual cues, such as an array of compatible web browsers' icons and a graphic of a generic URL bar containing the Transmission Control Protocol/Internet Protocol (TCP/IP) address assigned to the device, may be employed to simplify the language used in the on-screen text.

Next, the user must interact with a top-level menu page that will provide access to the various interfaces provided by the UI. As this use case is the first task a student is likely to perform with the interface, a visual cue highlighting the "*Tutorials*" link and icon will be used when a session is started in order to guide new users.

Descriptive labels for fields in the tutorial page's forms and a sensible split-screen layout will mesh with existing mental models. Similar document layouts are often used when students write school papers, with word processors on one half of the screen and web browsers or document viewers on the other. More icons will also be employed to carry over the embedded editor's model to the main IDE page.

In Use Case 3.2, an assumption is made that the Tutorial use case has been completed. If the student is intending to develop his own Verilog module, he has already accessed the "*Tutorials*" page and, consequently, is already connected through his web browser.

Link styles will be consistent across the entire menu page. As such, the programming language selection method must adhere to the page's style, while still being self-explanatory. Tactics such as integrating the drop-down menu into the link's text, or using it to modify the link's icon, rather than including it as a separate field may be used to accomplish a coherent User eXperience (UX).

Carrying over visual elements from the tutorial page's editor frame will help a student to quickly identify the various components required to write, compile, debug, and run his code.

The task of retrieving support documentation, as outlined in Use Case 3.3, must attempt to adhere to the mental model developed by the previous use cases. The tutorial and IDE pages employ a single-page design strategy. As such, the "*Help*" page should likewise follow a single-page strategy. The

initial state of this UI has a search bar above an index of document categories. Once the student performs an action, whether it be conducting a search or selecting a section of the index, the results will appear below the search bar and the index will move to the upper right-hand corner. Following a link to an internal document will display the contents in the same way, with both the search bar and index visible at the top of the page.

The help page’s visual representation on the main menu should be consistent with those of the other elements but should use a positional or chromatic variation in order to draw the student’s eye and ensure that its location is well known. This is necessary to ensure that users are confident knowing that assistance will be provided in the event that the platform fails to perform as expected.

In addition to the cues described above, the main menu’s hyperlinks will be enhanced with descriptive tags that will pop up when the student’s cursor hovers over them. This active feedback will increase the likelihood that the correct steps are performed to ensure task success.

Use Case 3.1: Tutorial Use Case

Use Case	Opening a Verilog Tutorial
<i>Description:</i>	A student has hooked up DEVBOX and will connect to the device to access a Verilog tutorial.
<i>Primary Actor:</i>	Student
<i>Preconditions:</i>	<ul style="list-style-type: none"> • Device is connected to display and WiFi network • Device power is on
<i>Postconditions:</i>	Selected tutorial is displayed in browser
<i>Main Success Scenario:</i>	

1. Student turns on display and sets its input to DEVBOX output
 2. DEVBOX IP address and connection instructions displayed
 3. Student opens browser on personal computing device (eg laptop, tablet) and enters IP into address bar
 4. Browser displays welcome/menu page
 5. Student clicks on “*Tutorials*” link
 6. “*Tutorials*” page loads
 7. Student selects *Verilog* from *Programming Languages* drop-down box
 8. Student selects desired tutorial from revealed list and clicks “*GO*”
 9. First page of selected tutorial appears in browser.
 - Instructional text in left-hand column
 - Editor with example code in right-hand column
 10. Student reads material, modifies code, and clicks *Compile & Run*
 11. Device executes code
 - (a) “*Messages*” textbox below editor displays warnings encountered during compilation
 - (b) “*Output*” textbox relays runtime messages from program execution
 - (c) Arduino/Verilog I/O visualizer appears on display
 - (d) Device I/O and display elements change to reflect Verilog emulation state transitions
 12. Student Clicks the *Next* button
 13. Next tutorial page is displayed and steps 10 to 13 are repeated until final page is reached
-

Use Case 3.2: Project Use Case

Use Case	Starting a Verilog Project
<i>Description</i>	A student begins and configures a new Verilog hardware design

Primary Actor: Student

Precondition: Student is accessing device's menu page

Postconditions:

- HDL code compiles and is executed with Verilator
- Port signals connect to hard I/Os and *clock* behaves correctly

Main Success Scenario:

1. Student selects *Verilog* from drop-down menu next to "*Editor*" link and clicks on link
 2. Editor screen is loaded in browser
 3. Student enters list of port signals in text field next labeled `module main`
 4. Using a series of drop down menus on Right-Hand Side (RHS), Student associates signals with hard I/Os and *clock*
Clock cycle count configured to be infinite or to terminate emulation after *N* cycles
 5. In main text box, student composes HDL
 6. Student clicks "*Compile and Run*"
 7. Device compiles code into executable and emulates design on back-end
 8. Compilation warnings and errors are displayed in "*Compile Messages*" text box
 9. *\$display* messages are printed in "*Program Output*" text box
 10. Hard I/Os respond during emulation according to described logic
-

Compile Failure Extension:

7-10a.

7. Device attempts to compile code
 8. Detailed error messages are displayed in "*Compile Messages*" text box
 9. Student corrects structural and syntactical errors described by error messages
 10. return to step 6
-

Use Case 3.3: Help Resources Use Case

Use Case	Accessing Support Documentation
-----------------	--

<i>Description</i>	A student encountering difficulty uses included “Help” documentation to resolve issue
--------------------	---

<i>Primary Actor:</i>	Student
-----------------------	---------

<i>Precondition:</i>	Student is accessing device’s menu page
----------------------	---

<i>Postcondition:</i>	Student is provided with sufficient information to continue work
-----------------------	--

Main Success Scenario

1. Student clicks on question mark icon on menu webpage
2. “Help” page displayed in browser
 - Search field at top of page
 - Help document categories listed below:
 - (a) Frequently Asked Questions (FAQ)
 - (b) Manual
 - (c) Programming language reference material
 - (d) Troubleshooting guide
 - (e) External links
 - (f) “Contact Us”
3. Student enters keywords into search field and clicks “Search”
4. List of matching document sections displayed with brief excerpts, sorted by relevance, similar in format to Google search results
5. Student clicks on link deemed most relevant
6. Document is displayed at heading of indicated section

Extensions:

1. (a) Access from other pages:
 1. Student follows “*help*” link from any other page within interface
 2. “*Help*” page is opened in a new tab or window
 - (b) Direct access:
 1. Student enters URL of “*help*” directly into browser
 2. “*Help*” page is displayed in browser
 - 3-6. (a) Accessing local documents:
 1. Student clicks on category containing local documents
 - Manual
 - Programming language reference material
 - Troubleshooting guide
 2. A list of documents in the given category is displayed with brief descriptions
 3. User clicks on desired document
 4. Browser displays contents of selected document
 - (b) Accessing external web content:
 1. Student clicks on category containing links to external webpages
 - External Links
 - “Contact Us”
 2. Page is displayed containing:
 - list of hyperlinks accompanied by brief descriptions
 - related or pertinent text, such as mailing address
 3. Student clicks on desired link
 4. Browser is redirected to selected page
 - 4a. Search returns no results
 - (a) “No results found” displayed in place of document sections
-

3.4 Device Hardware

The proposed device for the DEVBOX platform is a simple television peripheral with a small form factor and low-profile I/O hardware. Figure 3.3 demonstrates three examples of existing products to which DEVBOX might bear resemblance. Figure 3.3a, for example, is Google’s Chromecast. It provides streaming video to a display device such as a television or computer monitor via HDMI. AppleTV, in Figure 3.3c, is much larger and performs a similar task as Chromecast but includes internal storage for user state and app functionality. The Compute Stick in Figure 3.3b, on the other hand, provides a miniature computer in a thumb drive-like device. All three of these devices share the common traits of small packaging, HDMI connectivity and minimal exposed I/O.

In order for the DEVBOX device to meet all of the platform’s requirements, certain hardware peripherals are necessary. For instance, without a processor, it would be unable to compile and run user code, or host the interface. Table 3.2 maps some of the more important requirements to hardware elements included in the design.

Table 3.2: Requirement-to-Component Mapping

Requirement	Component
Remote user access	WiFi adapter with WPS
Native compilation and execution Web hosting	ARM processor
Video feedback	HDMI controller
Multiple HW paradigms HDL simulation Platform flexibility Ongoing development	FPGA
Environment modularity	microSD-based filesystem
Peripheral compatibility	Arduino shield GPIO
On-board user I/O	Switches, buttons and LEDs
Serial developer connection	MicroUSB slave controller

3.5 User Interface

The primary UI is delivered by way of an internal web server. Students connect to the device via a web browser, and the WiFi-connected DEVBOX presents visual elements through the browser to guide users through the learning process. This method of delivery was selected to allow for a platform-

agnostic IDE. Students are able to connect to the device with virtually any web browser on any computer connected to the local network. This includes, but is not limited to:

- Windows PCs
- MacOS PCs
- Linux PCs
- smartphones
- tablets
- Mozilla Firefox
- Google Chrome
- Apple Safari
- Microsoft Edge

Figure 3.4 is a proposed layout of the various UI elements for DEVBOX. “*Tutorials*” provides a dynamic interface in which, as the student progresses through tutorial’s pages, an integrated code editor modifies the example code to reflect the new page’s subject matter. Users are able to review, modify, compile, and run example code without changing windows, leading to a more fluid learning process. “*IDE*” provides a bare-bones development environment in which users can build their own programs or modify project templates and then compile, run, and debug them. Because of major differences between Verilog and software languages, there are special I/O-related interface elements to help configure the code to work with DEVBOX’s hardware. Verilog modules use ports to connect to one another and to I/Os. The ports of the top-most module must be mapped to the hard I/O devices of the system in order to be observed by students. “*Project Templates*” provides a library of skeleton and vanilla example programs upon which students can build. “*Project Wizard*” provides a step-by-step guide to commenting and structuring a new project. “*Language Documentation*” contains free-license manuals and textbooks for the various languages provided by DEVBOX. These are stored locally so Internet access is not a requirement. Finally, “*Help Files*” provides troubleshooting and FAQ documentation specific to DEVBOX.

3.6 Device Software

There are many software elements within the DEVBOX device platform. Each one plays a different role in delivering the learning and programming environment to its students. Figure 3.5 shows their

organization. The top element, the Operating System (OS), provides a layer of abstraction and security as well as vital hardware support to the other applications. Although students never interact with the OS itself, it enables all of the tools that DEVBOX provides.

Within the OS are three vital elements. The first is the HyperText Transfer Protocol (HTTP) daemon. It delivers the primary user interface in the form of a dynamic web application. The front end of this website is primarily written in HyperText Markup Language (HTML) but also relies on Cascading Style Sheets (CSSes) and JavaScript (JS) for many client-side features and activities, such as interactive menu systems, contextual code highlighting, and live program feedback. Back-end elements of the web app are provided with PHP: Hypertext Preprocessor (PHP)¹ scripts. These scripts modify the appearance and structure of existing pages based on form input data. They are also involved in some file system operations required for successful compilation of users' code. The final component of DEVBOX's HTTP service is the Socket server. This Node.js host application performs the compilation and execution of students' programs independent of HTML preprocessing.

This software configuration loosely resembles the Asynchronous JavaScript and XML (AJAX) web development platform, wherein client-side elements communicate asynchronously with the server.

Next to the HTTP server several user-space applications provide valuable services, foremost of which are the compiler toolchains. The GNU C Compiler (GCC), for example, is a set of tools that transforms C and C++ code into binary applications. For each language that the platform supports, a different compiler may be required. Certain scripting languages, such as JavaScript and Perl, do not require compilers. They instead have parsers and interpreters that execute each command at runtime.

In the case of Verilog, the code must be synthesized in order to be simulated in an application like ModelSim, or assembled into a bitstream to configure FPGAs. This multi-stage process is complex and lengthy. Synthesis into an FPGA can take from 5 minutes to multiple hours depending on design complexity. For the purpose of training, however, most tutorial-based designs are simple enough to be quick to synthesize.

Commercial synthesis tools such as Altera's Quartus II are closed-source and not compatible with ARM processors. Instead, open-source HDL emulators, such as Verilator, can be ported and used

¹PHP is a recursive acronym which means that it contains itself. Another example of this is GNU, which stands for "GNU is Not Unix".

to parse Verilog and generate equivalent C++ code. This code is then compiled into a stand-alone software application and run.

Certain languages require special software in order to be executed, called Run-Time Environments (RTEs). Java is one example of an RTE-dependent language. The benefit of using an RTE for program execution is relative portability. The compilation process for developer applications can be cross-compatible. A Java application compiled on an x86 Windows PC will also run on ARM Linux, assuming the native RTE has been installed. In the case of Arduino C, program execution is heavily dependent the ATmega microcontroller's architecture and its connected I/Os. As such, it also requires an RTE in order for DEVBOX to simulate the behaviour of the Arduino board's hardware.

The HTTP server and user applications are vital, but without access to the device's hardware and platform-specific libraries students will not be able to compile their programs. Because DEVBOX uses a custom hardware configuration and architecture, specialized device drivers are required to expose hardware I/Os to the students. Similarly, software libraries with human-readable APIs are necessary to facilitate usability.

3.7 Third-Party Development

Section 3.2 identifies educators as a user group for the DEVBOX platform. In order to facilitate the addition of programming languages and instructional material into the system, Certain required activities will be standardized or unified. For instance, the use of a centralized `Makefile` and language description table such as `langtable.csv` (Table 3.3) drastically reduces the number of documents the user must edit in order to add a new programming language and compilation/execution script to the platform. In this document, developers identify the display name for the language, the script used by the IDE for contextual highlighting, its Multipurpose Internet Mail Extension (MIME), or Internet media, type, and the name of the file to which student code will be written. The web server uses this information to populate drop down menus, configure editor text and correctly compile student code.

Adding tutorials is also simplified by dynamically generating the tutorial list within the web interface by parsing a standardized directory structure within the device's file system. As long as developers add their custom tutorials to the disk image following the simple prescribed format, illustrated in Figure 3.6, the interface will present them correctly.

Table 3.3: Langtable Example

Language	JS Script	MIME type	Writeback File
c	codemirror-4.2/mode/clike/clike.js	text/x-csrc	main.c
arduino	codemirror-4.2/mode/clike/clike.js	text/x-csrc	main.c
verilog	codemirror-4.2/mode/verilog/verilog.js	text/x-verilog	main.v
vb	codemirror-4.2/mode/vb/vb.js	text/x-vb	main.vb
java	codemirror-4.2/mode/clike/clike.js	text/x-java	main.java
LaTeX	codemirror-4.2/mode/stex/stex.js	text/x-stex	main.tex

The most challenging part of contributing a new programming language will be cross-compiling the compiler tools. In certain cases, as with GCC, configuration scripts and tools are provided in order to facilitate this cross-cross compilation. The configuration tools for other compilers may not provide settings for ARM-native builds.

Figure 3.3: Form-Factor Analogs



(a) Google's Chromecast (image www.google.com)



(b) Intel's Compute Stick (image www.intel.com)



(c) Apple TV (image www.apple.com)

Figure 3.4: User Interface Elements

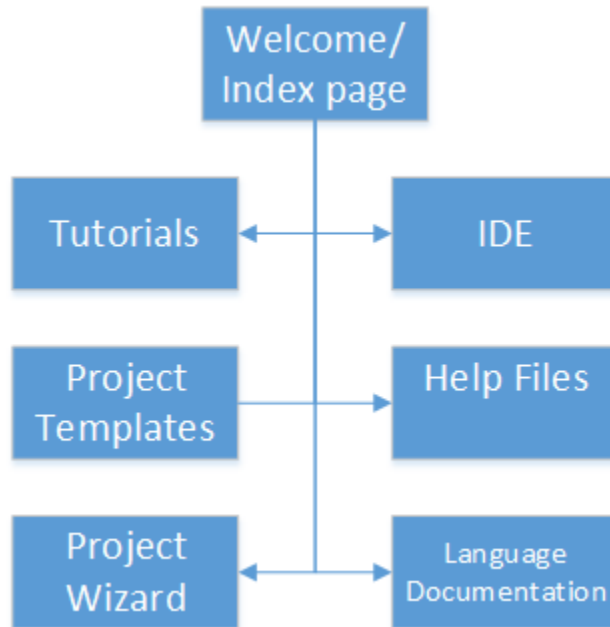


Figure 3.5: DEVBOX Software Component Hierarchy

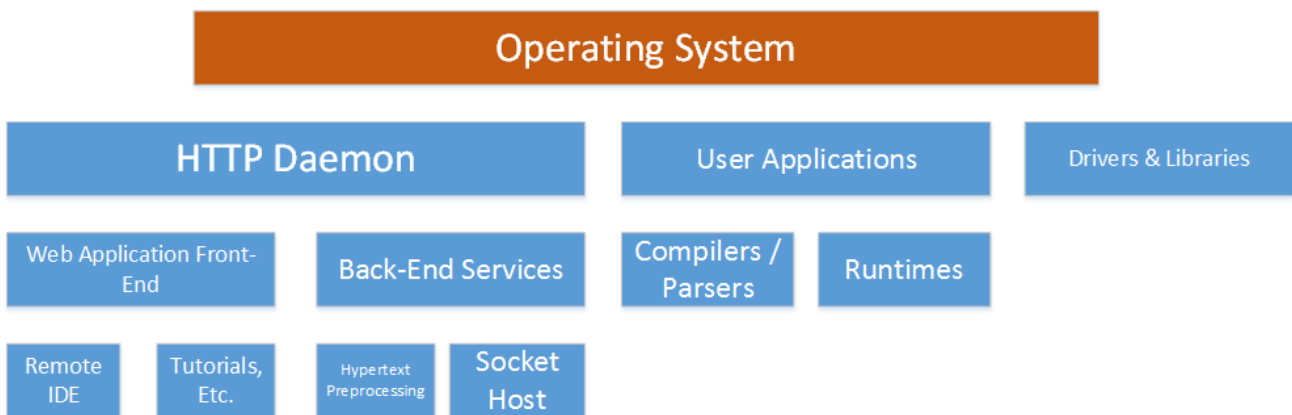
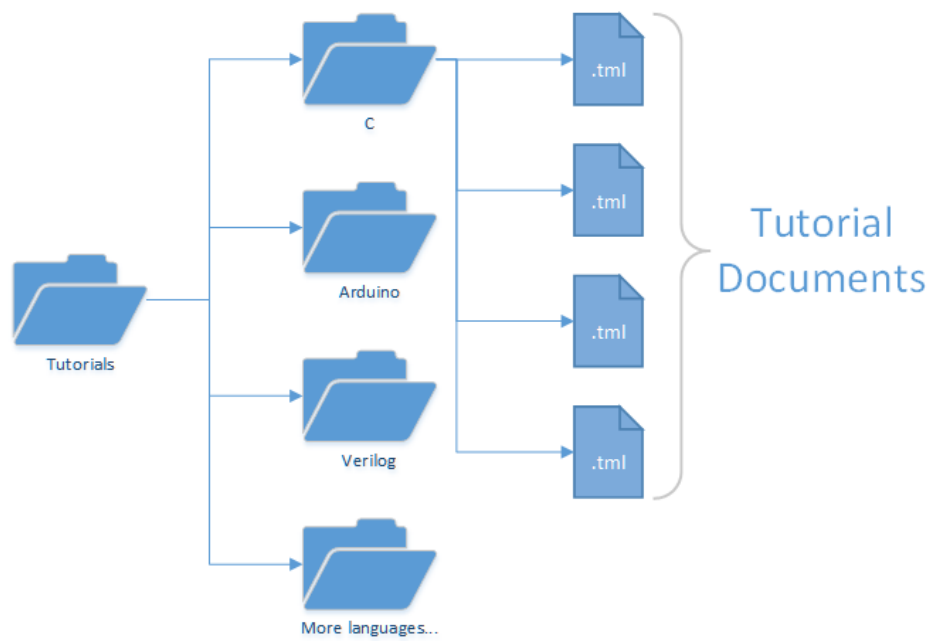


Figure 3.6: Standardized Tutorial File Structure



Chapter 4

The DEVBOX Prototype

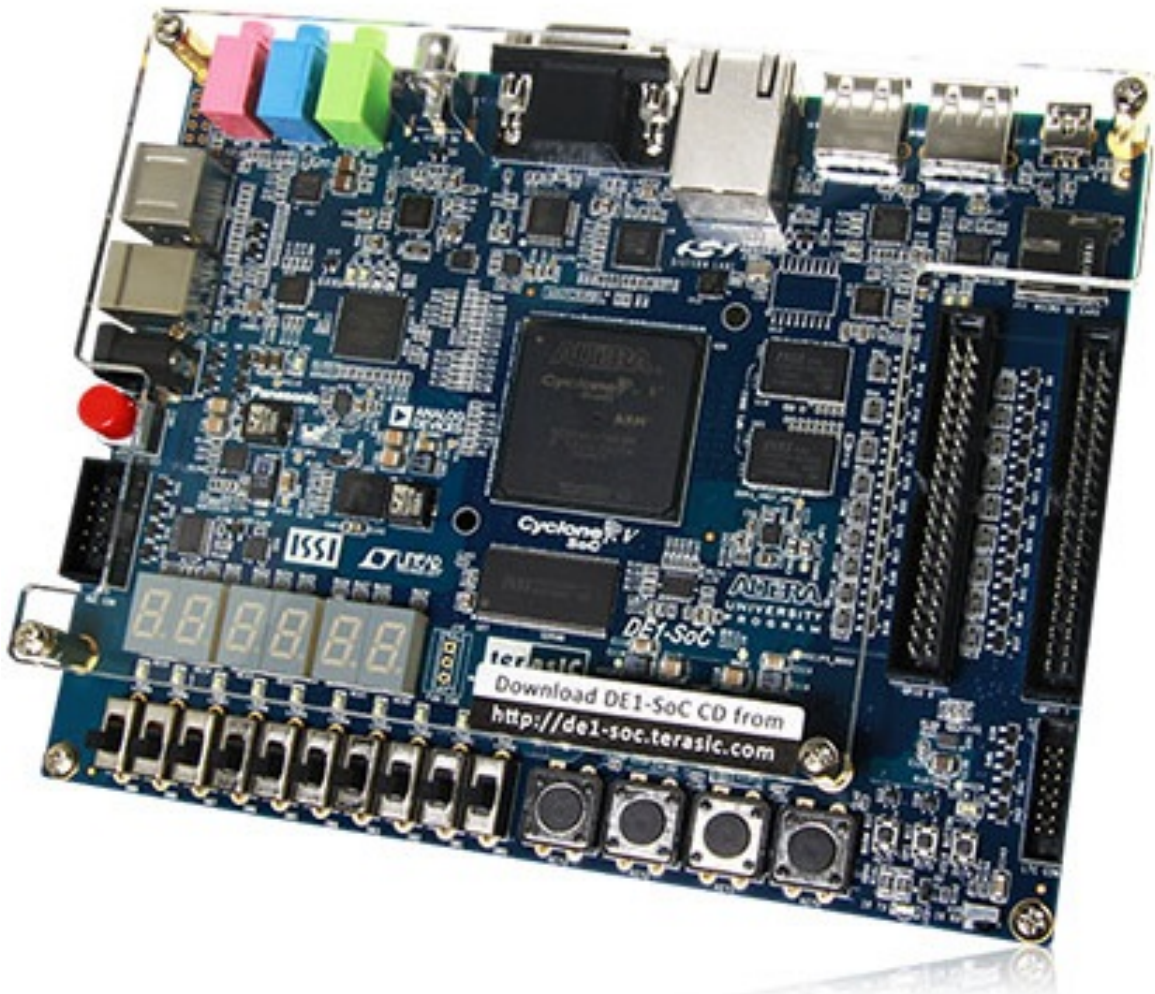
The DEVBOX prototype was developed to demonstrate the ability to provide the tools and resources necessary for software and hardware programming education in a single embedded device as described in Chapter 3. In order to accomplish this task an existing, flexible SoC-FPGA hardware platform was chosen. Its software and FPGA configuration was customized to DEVBOX's specific needs.

In this chapter the elements developed or employed to produce the functional prototype are discussed. Section 4.1 describes the device upon which DEVBOX was built. Section 4.2 outlines the FPGA hardware elements used by the prototype and how the device accesses them. In Section 4.3, the OS and the libraries it uses to deliver the prototype's services are discussed. The deployment of the browser-based mock-up UI and the tools that manage it is covered by Section 4.4. Section 4.5 provides details concerning the development of custom device drivers required to expose the Cyclone V's FPGA hardware and Section 4.6 describes the required applications and tools for compiling and running code with the DEVBOX prototype.

4.1 Device

In order to implement the DEVBOX prototype, an embedded computer and OS as well as a connection to a local network and to a video display are considered to be the basic minimum requirements for the prototype device. For this and a great many other reasons, Terasic's DE-1 SoC FPGA development board, as seen in Figure 4.1, was selected.

Figure 4.1: The Terasic DE-1 SoC Board (image: www.terasic.com.tw)



An SoC FPGA includes, alongside the FPGA hardware, a Hard Processing System (HPS). This is a physical CPU and its fundamental satellite components, including the Memory Management Unit (MMU) and data bus. The inclusion of a fast on-board processor with support for virtual memory facilitates the addition of a fully operational Linux kernel.

Terasic Development and Education (DE) boards are often used in ECE programs at many universities, including The University of British Columbia (UBC). The tools provided by Altera for these devices are well documented and familiar to many students in ECE programs.

The board, shown in Figure 4.1, is built around Altera's Cyclone V-SoC FPGA chip[14]. As shown in Table 4.1, The processor is a dual-core Cortex-A9 and is one of the fastest ARM designs to

Table 4.1: ARM Cortex A9 Technical Specifications

Spec	Value
Clock Speed	800 MHz
# Cores	2
Instruction Cache	64 Kilobyte (KB)
Data Cache	64 KB
L2 Cache	256 KB
RAM	1.0 Gigabyte (GB) DDR3
Storage	Micro SD card socket
Communication	Gigabyte Ethernet
Virutal Memory System	ARM MMU

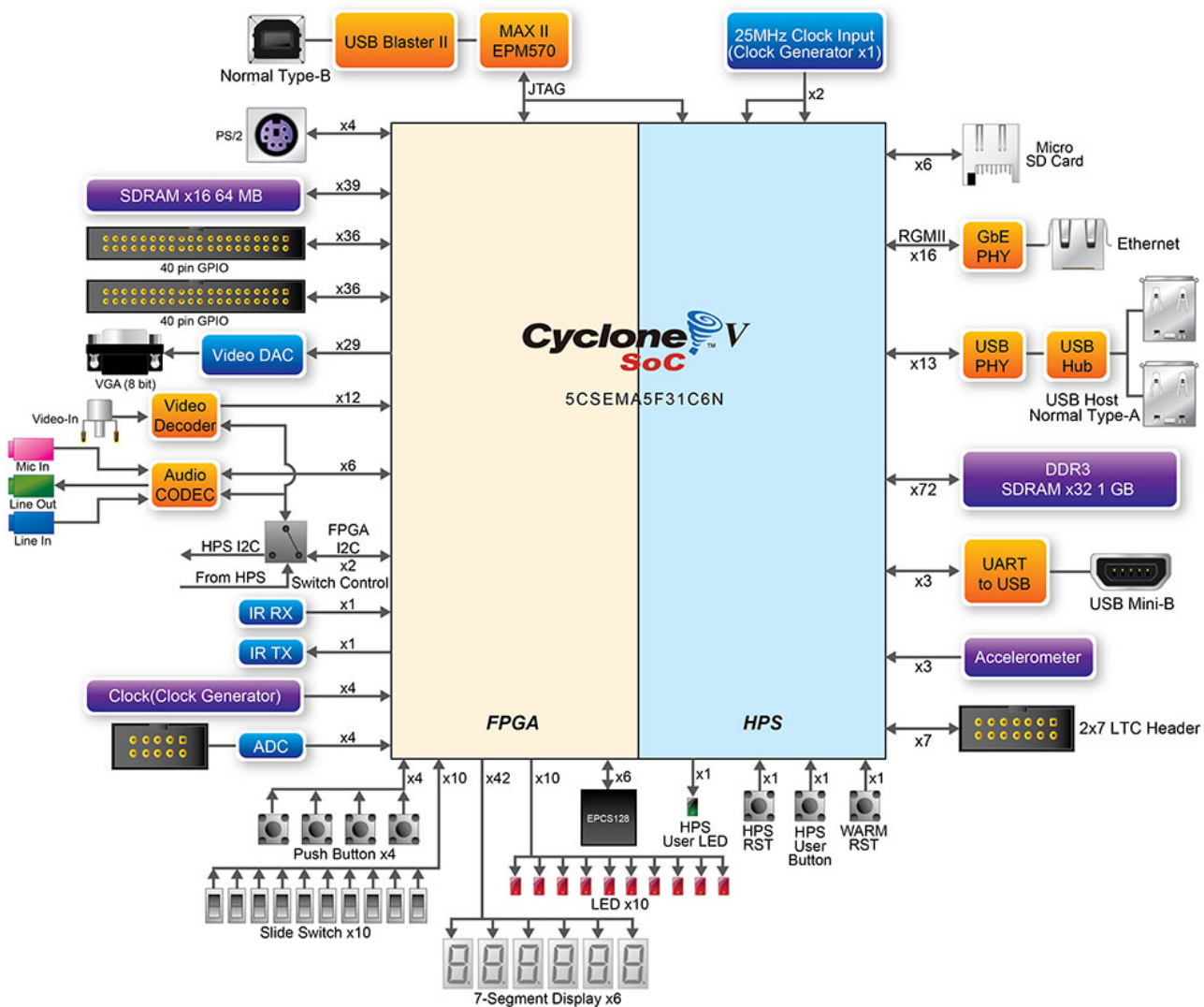
date, running at 800 Megahertz (MHz). The system contains 1GB of third-generation third-generation Double Data Rate (DDR3) Random Access Memory (RAM), enough to host a full Linux environment. Figure 4.2 shows the layout of the device's various I/Os with respect to the HPS and FPGA regions of the chip. Beyond the required Video Graphics Adaptor (VGA) DAC hardware and RJ45 Ethernet controller, the DE-1 SoC hosts a wide selection of I/O devices

By combining this device with a VGA monitor and an Ethernet connection to a WiFi, router the DEVBOX test workstation (Figure 4.3) was set up in a lab environment. Beyond these two external devices and a power cord, no other physical connection to the DE-1 was required during normal development and operation. The one exception to this is that modifying the bootloader's environment variables and execution parameters a requires physical connection to the HPS's microUSB Universal Asynchronous Reciever/Transmitter (UART) interface.

4.2 FPGA Hardware and I/O

Many useful and important hardware devices are connected to the FPGA region of the DE-1 SoC's Cyclone V chip. In order for the HPS to access them, the FPGA must be programmed with hardware controllers for every desired I/O. These modules map the pins connected to external elements to combinational and sequential logic that produce correct device behaviour. They, in turn, connect to internal communication channels to and from the HPS or between hardware controllers. Modern ARM processors use a communication protocol and architecture known as Advanced eXtensible Interface (AXI) [13]. Each entity on the AXI bus is defined as either a *master* or a *slave*, where

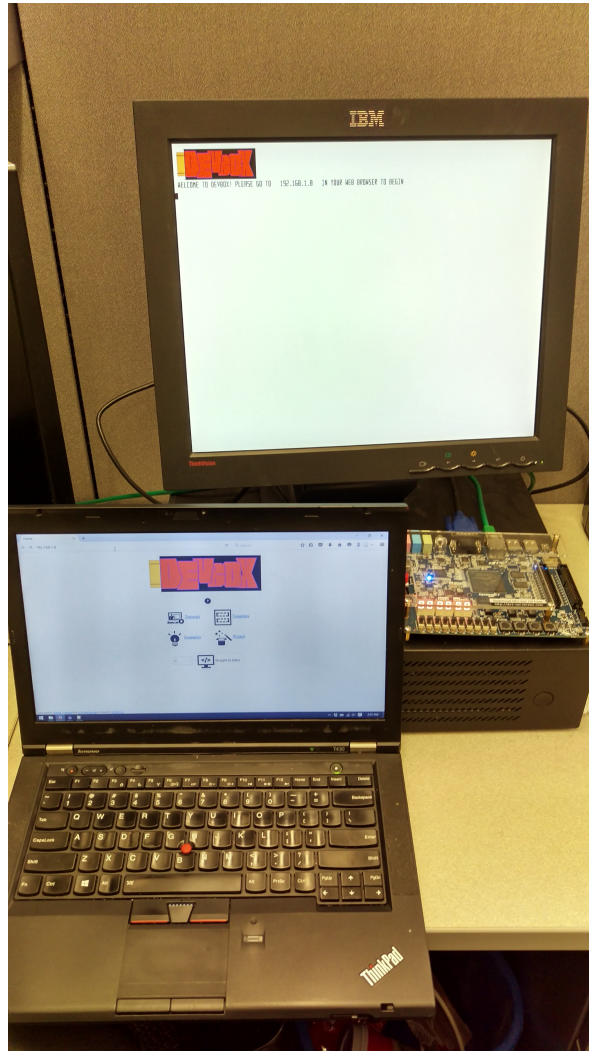
Figure 4.2: DE-1 SoC Layout (image: www.terasic.com.tw)



masters issue instructions to be carried out by slaves. Figure 4.4 demonstrates how the ARM HPS is connected to the various hardware modules in DEVBOX's FPGA design. Most of the clock and reset signals are omitted for clarity. AXI and Altera's Avalon bus interfaces are strikingly similar and completely compatible. The HPS can easily communicate with Altera's pre-designed hardware Intellectual Property (IP) cores without intermediary hardware.

Each of these modules is assigned a physical address to which the HPS can refer. In the DEVBOX FPGA design, most modules are either Memory Mapped (MM) master devices initiating read

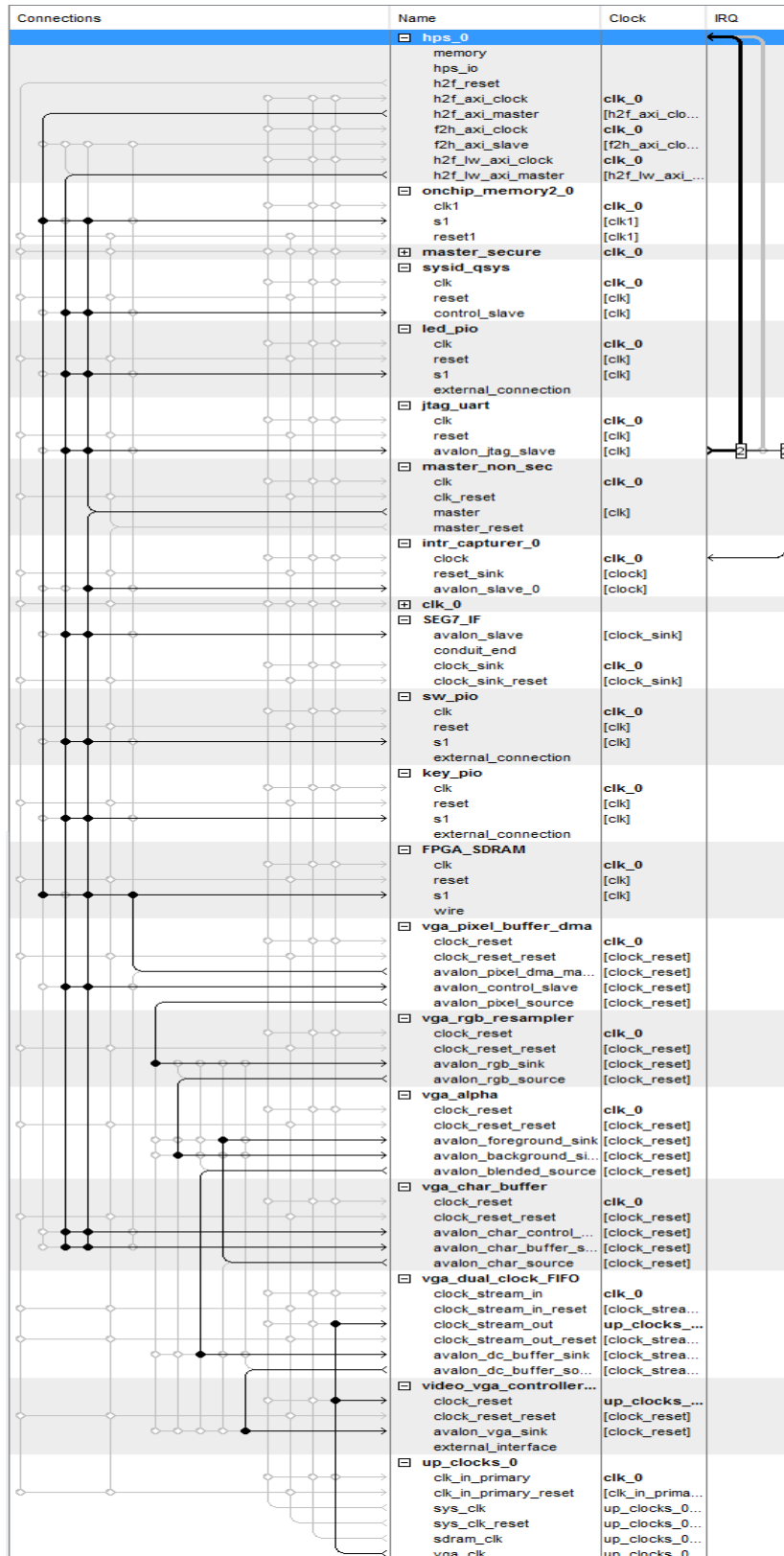
Figure 4.3: DEVBOX Prototype Workstation



and write requests or MM slaves fulfilling these requests. There is also a streaming communication interface. Avalon Streaming (Avalon-ST) is often used between components that send or receive large amounts of data, such as or Ethernet controllers. For MM communication between the ARM processor and the FPGA, a request is sent along the AXI bus from the HPS, including the address of a MM slave. This request is transformed and issued along Altera's Avalon architecture and received by the target module. Some of these devices, such as the VGA controller, are intricate and require multiple hardware components to implement.

As is evident from Figure 4.2, the VGA hardware is connected to the FPGA and not to the HPS.

Figure 4.4: DEVBOX Prototype FPGA System Diagram



It is therefore necessary to provide the VGA controller within the FPGA framework. Altera provides many pre-built and configurable hardware modules, commonly referred to as IP cores, or “soft” devices, to emulate common devices including VGA controller hardware. Without this infrastructure in place the DEVBOX prototype is unable to provide visual feedback to its users. The VGA control pipeline is laid out in Figure 4.5. It is important to have the external visual feedback of a separate display so as not to disrupt or detract from the student’s access to the UI. This allows students to continue editing code as they observe program behaviour.

First, the ARM HPS is connected to the pixel buffer Direct Memory Access (DMA) controller. This device streams data from the FPGA’s off-chip Synchronous Dynamic RAM (SDRAM) starting at a hard-coded address. This region of memory acts as a framebuffer for the OS. The pixel data from RAM is fed into the Red-Green-Blue (RGB) resampler to scale values from 32-bit RGB-Alpha (RGBA) to 30-bit RGB. The result of this pixel scaling is then sent, along with character buffer data, to the alpha blender. This mechanism combines the characters with the pixel data from the resampler, overlaying printed text on top of the framebuffer’s image.

At this point the video data must cross clock boundaries in order to be received by the VGA controller at a prescribed scan rate. The VGA controller then transmits the pixel data to the VGA Digital-to-Analog Converter (DAC) to be transmitted to the monitor.

The controllers for the other FPGA I/Os used by the DEVBOX prototype – Light Emitting Diodes (LEDs), switches, pushbuttons, and 7-segment numerical displays – are built using IP blocks in a similar way to the VGA pipeline, but are much simpler to deploy. With these IP blocks in place, the HPS has direct access to FPGA hardware.

4.3 Operating System and Drivers

The DEVBOX prototype’s software package is built upon a custom Linux kernel provided by Terasic. Due to the highly varied configurations of embedded systems in which these ARM processors are used, there is no universal Linux distribution available for them. Terasic has gone through the trouble of building a DE-1-compatible distro with only one important omission – it did not include the necessary libraries for compiling kernel modules. Linux Kernel Modules (LKMs) are required to provide user access to hardware I/Os by bridging the gap between unprotected user space addresses and the kernel’s

Figure 4.5: VGA Controller IPs

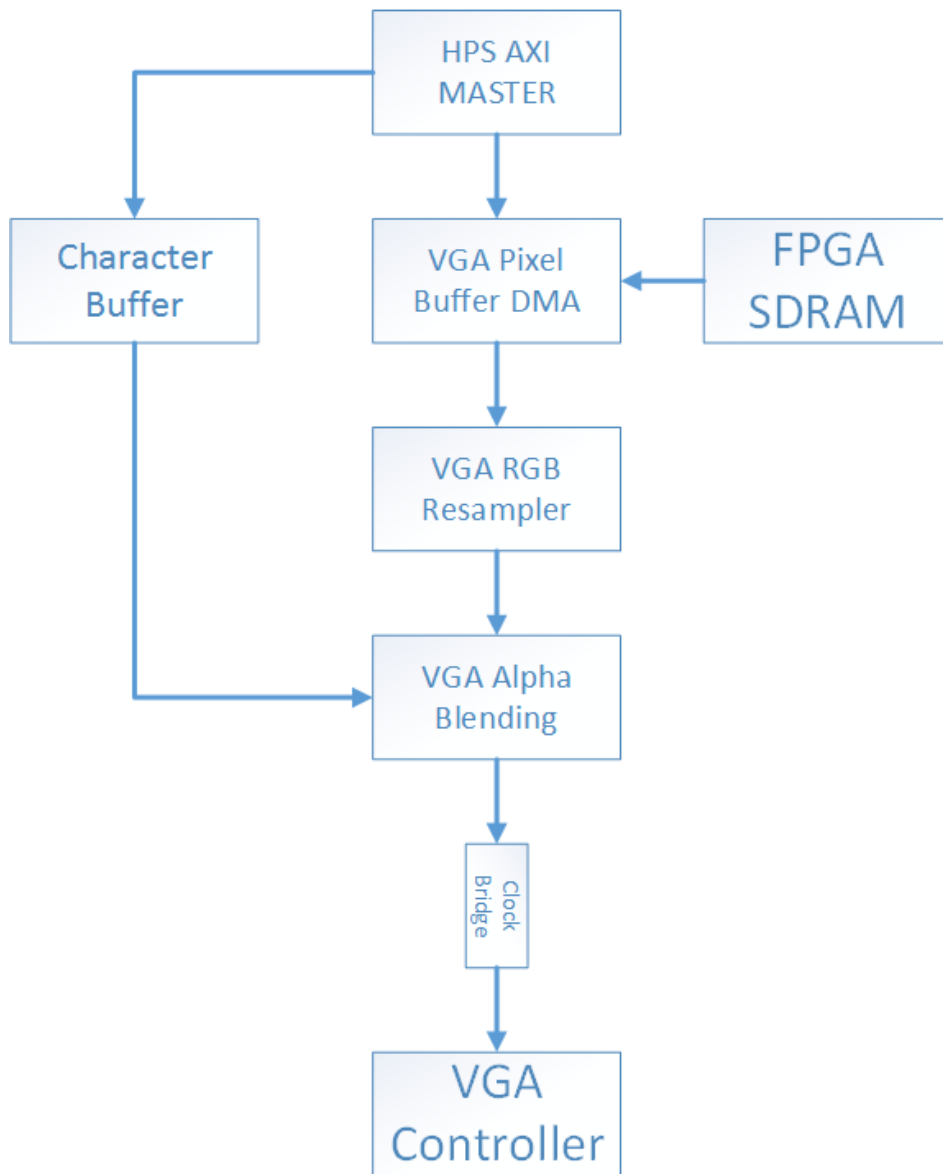
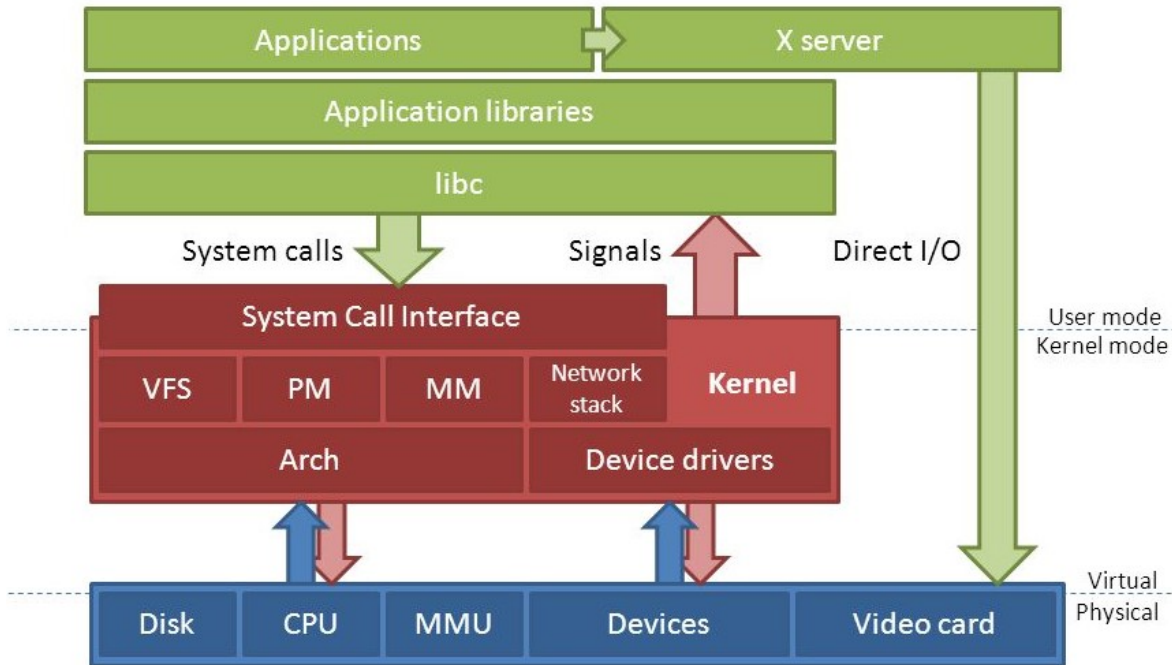


Figure 4.6: User Space Versus Kernel Space



closely guarded hardware addresses. Figure 4.6 is a generic outline of how user applications access resources protected by the Linux kernel.

User applications make system calls to the kernel, requesting the use of a physical resource or device. The resource’s LKM – also known as a device driver – sends the instruction associated with the request to the physical address of the device. The kernel receives the response data and passes it on to the driver, which then delivers it to the user application, and execution continues in user space. Table 4.2 is a partial listing of the physical addresses for the DEVBOX I/Os.

Terasic’s Linux distro includes a device driver for the soft VGA controller described in Figure 4.5 to provide an identical framebuffer device similar to those on standard Personal Computers (PCs), but there is no driver for the additional hardware added to the FPGA for the DEVBOX prototype. In order to provide user space access to these devices, a driver – a custom LKM paired with an API library – was developed. APIs provide software developers with simple, logically named and easy to use functions to perform complex or convoluted tasks such as driver access. The interaction between API and LKM is illustrated in Figure 4.7. The term “Character Device” refers to the method of

Figure 4.7: Character Device Driver Software Structure

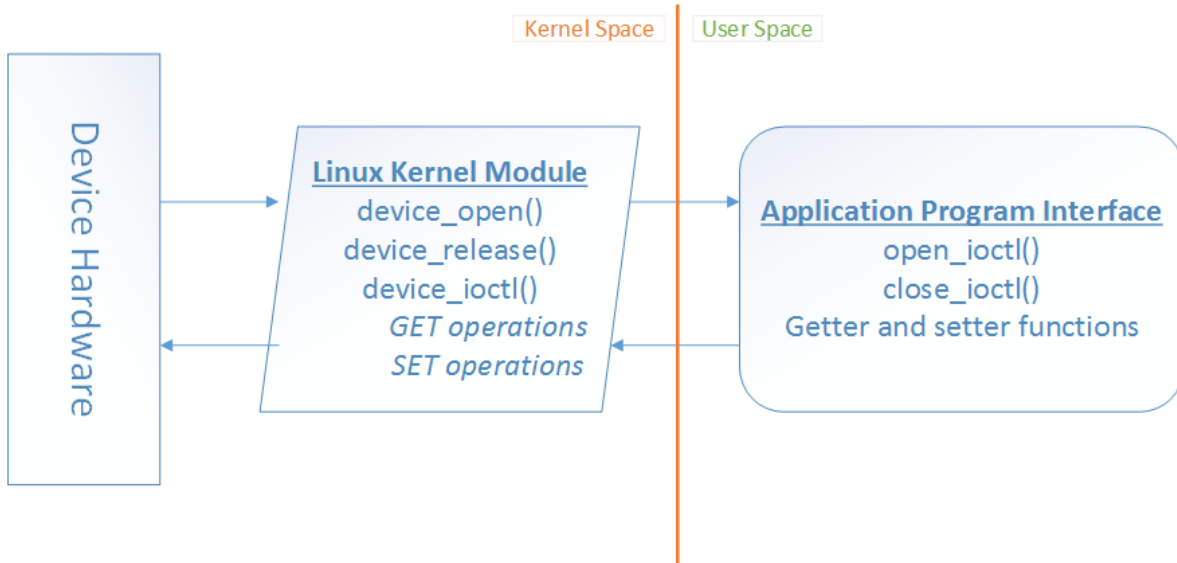


Table 4.2: DEVBOX Physical Address Map

Device	Description	Address(Hex)
LEDR	Red LEDs	0x00010000
KEY	Pushbuttons	0x00010010
SW	Switches	0x00010040
SEG7	7-Segment Displays	0x00010060

communication the driver is configured to use, accepting plain text as function arguments.

Compiling `devbox_ioctl`, the custom driver described above, required the “*modules*” libraries for Terasic’s Linux kernel. In order to supply them, the OS was rebuilt from the Yocto Project [12] repository. Yocto Project is a collaboration between hardware developers whose goal is to provide the tools necessary to build Linux distros for many commercial embedded devices, including Altera FPGAs. Once these modules are built and included in the `devbox_ioctl` compilation script, the resulting kernel object and the API library and header file are added to the DEVBOX file structure. The driver can then be registered with the kernel and accessed by user applications.

The registration process, involving module installation and node generation, places a user-accessible character device file in the OS’s `/dev` directory. A start-up script performs this task as well as configuring permissions for the `devbox_ioctl` node and framebuffer to provide read/write access to all users.

4.4 Web Server and User Interface

The previous sections of this chapter have described all of the necessary elements that support the prototypical DEVBOX platform. At this point, a UI for accessing these resources has not been discussed. The method selected for the DEVBOX prototype to provide the front-end interface was by way of a web application. These dynamic websites are relatively platform agnostic, as they depend only on the computer's web browser and not on the hardware or OS of the computer accessing them. Students will be able to use whichever computer they chose, such as a PC, Macbook, tablet or smartphone.

To this end, an HTTP daemon was configured to deliver web content to students across a local network connection. *Lighttpd*, a small, serviceable HTTP daemon was included with the original image and, with minor adjustments to its configuration, it proved capable of delivering the web services required by the UI.

In addition to the visible web content, several background activities must occur asynchronously from users' browser actions. Therefore, parallel to the HTTP daemon, a Node.js [15] "socket" server was developed to support back-end operations. The Node API enables JS embedded in webpages to trigger server-side activities through sockets – virtual data channels that connect devices over the Internet.

With the infrastructure for dynamic web applications in place, a mock-up UI was developed to demonstrate the DEVBOX concept. This functional prototype, built primarily with PHP, presents the interaction model for the major elements of the platform concept, namely tutorials and the IDE. General layout options for the less critical component are also explored, but as they do not pertain directly to the self-directed learning elements of the platform, their functionality was not explored.

Web applications are divided into two parts – the front-end and back-end. The front-end component consists of the visual elements displayed in the browser and processing tasks performed locally. The back-end preprocesses HTML content based on submitted form data and client state and performs any server-side computation.

4.4.1 The Back-End Code

The back-end, or server side, components of the prototype’s UI are the PHP preprocessing scripts and Node.js server. These two processes operate concurrently on very different tasks. Their use and implementation details are described here.

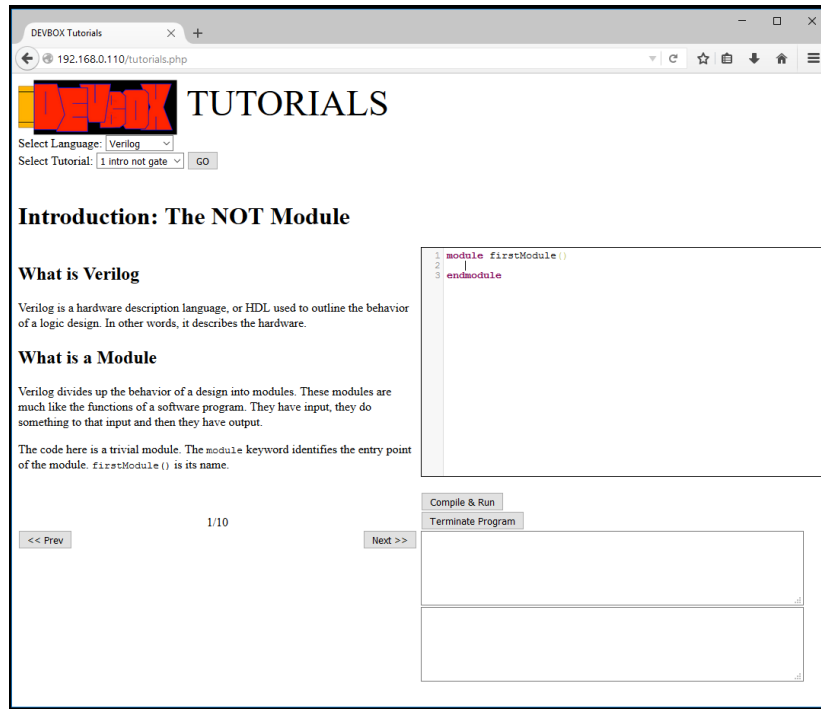
The PHP Preprocessor

Web applications are inherently stateless. Preprocessing allows the HTTP server to manipulate the content presented based on the data found in form submissions. These are received in the form of GET or POST requests transmitted by the web client when the user submits a form. There are often a large number of form fields, both hidden and visible, used to maintain user state within the DEVBOX interface; GET values appear in the site’s address bar so in order to minimize student confusion, form data is transmitted to the server with POST requests.

The PHP interpreter then regenerates the site’s HTML based on the new set of POST form variables. Listing 4.1 is an excerpt from the preamble for the “*Tutorials*” page, seen in Figure 4.8. Every time the client browser transmits a request to the server, the values of certain elements in the `$_POST[]` array are analyzed in order to load file contents, hide or expose elements, and populate specific regions of the HTML code before a response is transmitted to the user.

In another portion of `tutorials.php` the `/www/pages/tutorials` directory tree is parsed by the PHP script in order to display the current selection of tutorials on the page. The preprocessor also accesses `langtable.csv`, described in Section 3.7, in order to populate the list of available programming languages, save user code, and include the correct contextual highlighting script for the interface’s editor text box.

Figure 4.8: The DEVBOX Tutorials Page



Listing 4.1: Code Snippet from tutorials.php

```
1 <!DOCTYPE html>  
2 <?php  
3     ini_set( 'include_path', 'phplibs/' );  
4     include "TutoriML.php";  
5     include "portnum.php";  
6     if (isset($_POST[ 'lang ' ]))  
7     {  
8         $lang=$_POST[ 'lang ' ];  
9         $dir="tutorials / ". $lang ;  
10        $files=array_slice ( scandir ( $dir ) ,2);  
11    }  
12    $page=$_POST[ 'page ' ];  
13    $tml=$_POST[ 'tutorial ' ];  
14    $file='main.txt';  
15    $log='php_log.csv';  
16    $date=new DateTime ();  
17    if (isset ($tml))  
18    {  
19
```

```

20     if (isset($_POST['code']))
21     {
22         $base=$_POST['code'];
23         file_put_contents($file, $base);
24     }
25     $title=getTitle($dir, $tml);
26     $n_pages=getPages($dir, $tml);
27     if (isset($_POST['submit']))
28     {
29         $code=getCode($dir, $tml, $page);
30         $base=htmlspecialchars($base);
31         fsetCode($file, $code);
32         $base=file_get_contents($file);
33         file_put_contents($log, 'Tutorial_start, '.date_timestamp_get($date).';'.PHP_EOL,
34             FILE_APPEND);

```

The Node.js Server

The Node.js server's purpose is to perform server-side tasks asynchronously from HTML preprocessing tasks in order to provide live data to the front-end. The primary activities that require this service are compilation and execution of the student's code. Appendix B shows the source code for the Node server.

JavaScript run on the client side accesses the server's listening port, It performs filesystem activities and executes console commands according to variables provided by the client connection. In this snippet, after a connection is open, the Node server sleeps until it receives a "Compile and Run" command from the client. It then saves the received code to a document on the server, executes the `Makefile` script for the given language, and compile-time error messages are transmitted to the client. If compilation is error-free, the resulting binary is run, sending program output to and receiving program input from the client application.

4.4.2 The Front-End Code

The front-end, or client side code, like the back-end, is divided into two components. These are the HTML and JavaScript code. HTML is used to align, typeset and render web content within the

Figure 4.9: DEVBOX Prototype Index Page



client browser window. HTML is static and immutable once received by the client and its level of interactivity is restrictive. As described in the previous subsection, the PHP preprocessor allows for changes to the code within pages as they are reloaded, but it cannot modify content once the document has been transmitted. DEVBOX's index page, as seen in Figure 4.9, is an example of a static HTML page.

JavaScript

Inline with the HTML transmitted to the client on interactive pages, several JavaScript functions provide additional interactivity and functionality, like the *Editor* page. The JavaScript in lines 231 to 274 of `tutorials.pdf`, listed in Appendix C, configures the editor box and manages a socket communication with the Node server to receive and deliver the dynamic elements of the page.

Table 4.3: TutoriML Tags

Tag	Function
<code>`title</code>	Line contains name of current tutorial
<code>`n_pages</code>	Total number pages in the document
<code>`start n</code>	Begin documentation section for page n
<code>`end n</code>	End of page n
<code>`codeset</code>	Reset the contents of the editor textbox to code in this section
<code>`codereplace</code>	Replace given code with new code provided
<code>`codeappend</code>	Add code to the end of the codebase
<code>`codeprepend</code>	Add code to beginning of the codebase
<code>` `</code>	End section

4.4.3 TutoriML

Programming tutorials commonly have two primary components: instructional documentation, and example code. In order to simplify the process of generating material for DEVBOX's *Tutorials* page, a custom file syntax was developed. For most instructional websites, it suffices to simply insert example code inline with the instructional material. For the split-screen interactive format used in DEVBOX, however, it becomes necessary to separate the code from the text. To simplify this process, the Tutorial Markdown Layer (TutoriML) file syntax was developed. Text is written in the popular Markdown format and code is stored verbatim. They are separated by tags that indicate the type and function of the contained data. A full list of TutoriML tags is found in Table 4.3. The parser is a PHP script included in the *Tutorials* page that scans the selected document, and, based on its tagging, inserts the provided data into the HTML code when a tutorial's page is accessed.

The use of a custom syntax with minimized complexity is meant to allow the author to focus on writing the material rather than on the tool used.

Listing 4.2: TutoriML Syntax Example

```
1 `title 1 – Introduction to C
2 `n_pages 13
3 `start 1
4 <sub>This tutorial addapted from "[Introduction to C](http://www.cprogramming.com/tutorial/c/lesson1
   .html)" by Alex Allain.</sub>
5
6
7 ##Intro to C
```

8 Every full C program begins inside a function called "main". A function is simply a collection of commands that do "something". The main function is always called when the program first executes. From main, we can call other functions, whether they be written by us or by others or use built-in language features. To access the standard functions that comes with your compiler (the program that interprets the code and turns it into a program), you need to include a header with the '#include' directive. What this does is effectively take everything in the header and paste it into your program. The code to the right compiles into a working program.

9

10 Try it out. Try changing the string in the 'printf' statement and see how it changes the program output.

11

```
12 ''  
13 'codeset  
14 #include <stdio.h>  
15 int main()  
16 {  
17     printf( "I am alive! Beware.\n" );  
18     return 0;  
19 }
```

20 ''

21 'end 1

22

23 'start 2

24 ##Program Elements

25 Let's look at the elements of the program. The '#include' is a "preprocessor" directive that tells the compiler to put code from the header called 'stdio.h' into our program before actually creating the executable. By including header files, you can gain access to many different functions—the 'printf' function, for example, are included in 'stdio.h'.

26

27 The next important line is 'int main()'. This line tells the compiler that there is a function named main, and that the function returns an integer, hence 'int'. The "curly braces" ('{' and '}') signal the beginning and end of functions and other code blocks. If you have programmed in Pascal, you will know them as 'BEGIN' and 'END'. Even if you haven't programmed in Pascal, this is a good way to think about their meaning.

28 ''

29 'codeset

30 #include <stdio.h>

31 int main()

32 {

```
33 }
34 ''
35 'end 2
```

4.5 Application I/O Interface

The `devbox_ioctl` LKM and API provide an avenue of communication with the custom FPGA I/O devices for software development purposes. Listing 4.3 provides an example of an application that makes use of the interface provided by the driver.

This application first attempts to open the character file associated with the `devbox_ioctl` LKM. This is done in the same way as a regular file is opened, since everything in Linux is represented as a file. Once the test application determines that the module's character file has been successfully opened, it queries the state of the DE-1's switches and sets the LEDs to the same value. It then sets each of the 7-segment displays to an arbitrary sequence of bits and then writes a hexadecimal value to them and closes the character file.

Listing 4.3: FPGA I/O Interface Code Example

```
1 // devbox_ioctl_test.c
2 // (c) DEVBOX 2015
3 #include "devbox_ioctl_driver.h"
4
5 int main()
6 {
7     // Acquire character device
8     int x=open_devbox_io();
9
10    if(x<0)
11        printf("FAIL!\n"); //Acquire failed
12
13    else
14    {
15        printf("SUCCESS %x\n",x);
16
17        int q=devbox_get_sw(x); // Get switch state
18        printf("Switches= %x\n",q);
19        devbox_set_led(x,q); // Light up LEDs
20
21        // Set 7-seg to a raw value
22        devbox_set_7seg(x,0,123);
23        devbox_set_7seg(x,1,21);
24        devbox_set_7seg(x,2,123);
25        devbox_set_7seg(x,3,123);
```

```

26
27     // Write hex value to 7-segs
28     devbox_7seg_write(x, 0xaf7701);
29     // Release device
30     close_devbox_io(x);
31 }
32 }

```

Although access to the framebuffer is made trivial by its integration into the OS, as described in Section 4.2, APIs available for drawing to the screen are cumbersome at best. A straightforward visual output library, involving direct access to `/dev/fb0`, the character device driver for the display, was developed for users to use with their programs. In Listing 4.4, the use of this `fbDraw` library is demonstrated. The resulting program displays a white splash screen with the DEVBOX logo mock-up and the text from command line arguments on the connected display. The image on the computer monitor in Figure 4.3 was drawn using this program, displaying the device's IP address and brief access instructions when the device has completed its boot sequence.

The nomenclature and usage of the functions in the `fbDraw` library are straightforward and intuitive, making them easier to remember and implement. Functions like `fb_fillScr()` and `fbBmp_draw()` are self-explanatory and require a minimal number of arguments.

Listing 4.4: Framebuffer Interface Code Example

```

1 // fbPrintArg.c
2 // (c) DEVBOX 2015
3 #include "fbDraw.h"
4 #include "fbBmp2fb.h"
5 #define MARGIN 5
6
7 int main(int argc, char** argv)
8 {
9     dev_fb fb;
10    bmp bm;
11    int i;
12    //load bitmap
13    if (fbBmp_openBmp("/usr/splash_logo.bmp", &bm))
14        printf("failed to open /usr/splash_logo.bmp\n");
15
16    pixel cursor;
17    cursor.x=10;
18    cursor.y=10;
19    //debug: Print fb status to console
20    printf("FB status: %d\n", fb_init(&fb));
21    //White background

```



```

22     fb_fillScr(&fb, 255, 255, 255);
23     //draw logo
24     fbBmp_draw(&fb, &bm, 1, cursor.x, cursor.y);
25
26     cursor.x=10;
27     cursor.y+=bm.info.height+MARGIN;
28     if(argc>=2)
29     {
30         for(i=1;i<argc;i++)
31         {
32             //Print argument text
33             fb_printStr(&fb, argv[i], &cursor, 12, 0, 0, 0);
34             //print arg text to console
35             printf("%s\n", argv[i]);
36             fb_printStr(&fb, " ", &cursor, 12, 0, 0, 0);
37         }
38     }
39     else
40         return -1;
41     return 0;
42 }

```

4.6 Compilers and Tools

The utility of DEVBOX hinges on the platform's ability to compile and execute users' code natively. This requires the build and runtime tools for each included language to be compiled for ARM processors, thereby necessitating the use of open-source products. The language list for the current prototype is very short: C, Arduino C, and Verilog. The deployment of each tool set is described below.

4.6.1 C and GCC

The most common open-source compiler for C and C++ is GCC. Others, such as *Clang* for LLVM [4], though not as common, are equally viable candidates for the DEVBOX project. GCC was selected in part for its build configuration flexibility.

Multiple variations of ARM processors exist and may use slightly different instruction sets or have access to different on-chip hardware, such as MMUs. ARM-native GCC compilers are not supplied in a precompiled binary, but can be generated using a build tool called `crosstool-NG` and, in most cases, a platform-specific configuration file can simplify the task. If no such file exists for the target architecture, there is a Graphical User Interface (GUI) included for manually configuring the build. The tool is then built in such a way that ARM-compatible binaries and libraries that compile software

for ARM are generated. The full suite of build tools includes:

- the C and C++ compilers,
- the object linker,
- the back-end assembler,
- the library archiver, and
- important library-building tools.

Only the first three are necessary for building executable applications, but the inclusion of the others simplifies the prototyping process during the prototype's development. These native library building tools allow for quick iteration on the prototype's various projects by minimizing the number of file transfers from the development machine to the DEVBOX device. Libraries can be compiled, archived and deployed natively as they are debugged.

The output of GCC's cross-compilation is added to the file system on the DEVBOX prototype and the location of the compiler's executable binaries is appended to the OS's `PATH` environment variable.

4.6.2 Arduino C Runtime

Arduino is a family of microcontroller-based devices for developing interactive and programmable "objects". They have a common C-based programming language focused on simplifying the control of the boards' I/Os. Arduino C is ideal for new developers with an interest in robotics, interactive devices and technological art forms due to its simple syntax and structure. The programming model for Arduino has been included in the DEVBOX prototype because it demonstrates the versatility of the platform, pursuant of its goal of delivering a wide variety of development learning tools spanning an extensive range of students' interests.

The structure of an Arduino program contains two primary functions:

- `setup()`
- `loop()`

`setup()` is used to initialize the I/O pins on the Arduino board as either digital input or output. Any other activities that are only performed once, such as initializing variables, are also placed in the `setup()` function.

The `loop()` function is repeated continuously while the device is on. This is the meat of an Arduino application. It is where the program responds to stimulus, driving I/O pins.

Listing 4.5 is a simple Arduino program that activates one of four LEDs dependent on the state of three input pins.

Listing 4.5: Arduino C Code Example

```
1 #include "arduinoRuntime.h"
2 void setup(void)
3 {
4     int i;
5     for(i=0;i<10;i++)
6         pinMode(i, INPUT);
7     for(i=10;i<14;i++)
8         pinMode(i, OUTPUT);
9 }
10
11 void loop(void)
12 {
13     int x=digitalRead(0);
14     x|=digitalRead(1);
15     x|=digitalRead(2);
16     printf("%d\n",x);
17     digitalWrite(10, x);
18
19     x=digitalRead(3);
20     x|=digitalRead(4);
21     x|=digitalRead(5);
22
23     digitalWrite(11, x);
24
25     x=digitalRead(6);
26     x|=digitalRead(7);
27     x|=digitalRead(8);
28
29     digitalWrite(12, x);
30
31     digitalWrite(13, digitalRead(9));
32 }
```

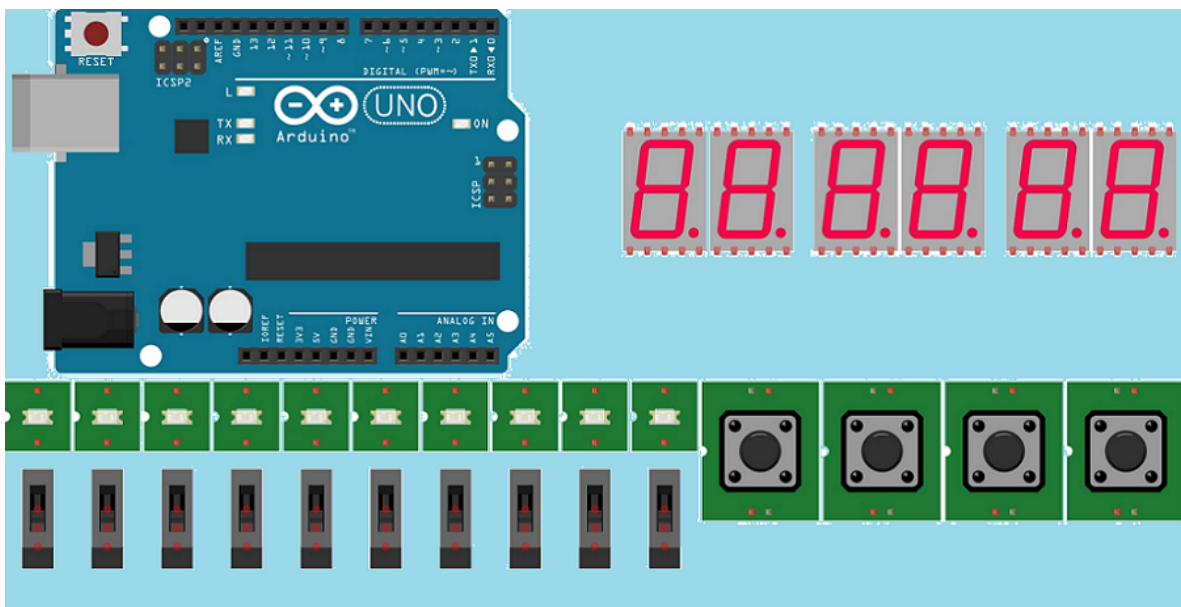
The major challenge behind integrating Arduino C into the DEVBOX platform stems from the simple fact that most Arduino boards are built around an ATmega microcontroller and are bare-wire

programmed. This means that there is no operating system between the code and the microcontroller. In order to emulate Arduino's behaviour on an ARM processor running Linux, a custom RTE, `ArduinoRuntime` was developed into which Arduino code could be compiled and run.

`ArduinoRuntime` consists of a porting library and an application into which the `setup()` and `loop()` functions are compiled. The porting library converts a large number of Arduino-specific functions to standard C and to the DEVBOX I/O interface. For instance, when `pinMode(i, INPUT)` is called, the function in the Arduino RTE associates the pin number `i` to the first available input device, `KEY[0]`, for instance. During `loop()`, if the state of that input is altered, the state of the emulated pin is changed accordingly by the RTE. A series of data arrays define the mode of each pin and associate the pin number with the board's I/O. The full Arduino RTE can be found in Appendix A.

The runtime application displays a graphical visualization of the Arduino Uno, one of the most common of the Arduinos, and the DE-1's pushbuttons, switches and LEDs on the connected video monitor. As the virtual I/O pins are read and modified, the graphical output highlights the active shield pins on image of the Uno and also reflects the state of the DE-1's hard I/Os, as seen in Figure 4.10. In this way, students are able to monitor the behaviour of an Arduino program in real time.

Figure 4.10: The Arduino Visualization Screen



4.6.3 Verilog Emulation and Verilator

Bare-Wire Verilog

A unique feature of the DEVBOX platform is the inclusion of educational materials for logic design and HDL education. Ideally, students' HDL would be fully synthesized into an FPGA bitstream and used to program the device's embedded FPGA. However, the tools required to do so are usually proprietary to the chip developer and not portable to ARM architectures. One alternative to this approach is to emulate the hardware model in software. Verilator [34] is an open-source development tool that parses Verilog code and generates equivalent C++ classes and functions. This software representation can then be compiled, along with a "harness" `main()` function. The harness is used to provide `stdin` and `stdout` communication and scripted I/O to manipulate the hardware emulation.

For the DEVBOX prototype, the harness application was designed to use the same I/O visualization used for `ArduinoRuntime`. Input and output wires defined in the Verilog modules are tied to DEVBOX I/Os and, as with Arduino, both the monitor and hard I/Os are manipulated to reflect the state of the emulation. In order to do this, a PHP script creates a custom Verilog file that instantiates the user-developed module and ties it to the port list that the Verilator harness expects. The modules are then converted into a C++ application and compiled into an executable binary.

The script for generating the harness module was written in PHP to facilitate its inclusion in DEVBOX's web based interface. The user defines their module's ports and manually ties them to I/O devices on the development board using the form in Figure 4.11 and when the user clicks the compile button the data from this form is handed off to the Verilog harness generation script. Listing 4.6 shows that the script builds Verilog code from scratch based on the signal list provided to the function by `POST` data from the web application. This standardizes the modules' external signal interface and simplifies the compilation process. The resulting Verilog code, shown in Listing 4.7, interacts with the main C++ function, which in turn interprets the `output` signals from the `harness` module and provides appropriate feedback from the host to the `input` ports.

The above emulator construction method is currently limited to simple designs involving only the switches, pushbuttons and red LEDs, but is fully integrated into the current UI mock-up. However, tools and methods are already in place for open-ended HDL development, such as a more generalized

Figure 4.11: Verilog Port Assignment Web Form

List your ports below:
module main(dipsw, pushbtn, clk);

Enter your verilog code:

```
1 //main.v
2
```

Emulation

Attach	Clock	to
clk		

OK

- attach SW dipsw
- attach KEY pushbtn
- clock set to count cycles: 20000

LKM, whose API is designed to mimic the core functions of Altera’s Avalon Hardware Abstraction Layer (HAL).

Listing 4.6: The create_v_harness.php script

```
1 <?php
2 function create_v_harness($sig_array)
3 {
4     \\ initialize variables
5     $assigned['SW']=0;
6     $assigned['KEY']=0;
7     $assigned['LEDR']=0;
8     $clock['set']=0;
9
10    \\ get signals from input array
11    foreach ($sig_array as $elem)
12    {
13        $elem=preg_split('/\s+/', $elem);
14        if($elem[0]=='attach')
15        {
16            $attach[]=array($elem[1], $elem[2]);
17            $assigned[$elem[1]]=1;
18        }
19        elseif($elem[0]=='clock')
20        {
21            $clock['set']=1;
22            if($elem[3]=='count')
23                $clock['count']=$elem[5];
24            else
25                $clock['count']=0;
26        }
27    }
28
29    \\ Verilator preamble to deactivate fatal errors for unused signals
30    $harness="//verilator lint_off UNUSED".PHP_EOL;
31
32    \\ Default top-level I/O signals
33    $harness.='module harness(CLK, CLK_EN, SW, KEY, LEDR, CLK_COUNT);'.
```

```

    ↪ PHP_EOL;
34 $harness.=' input wire CLK;'.PHP_EOL.' input wire[9:0] SW;'.PHP_EOL.'
    ↪ input wire[3:0] KEY;'.PHP_EOL;
35 $harness.=' output wire CLK_EN;'.PHP_EOL.' output wire[9:0] LEDR;'.
    ↪ PHP_EOL;
36 $harness.=' assign CLK_EN='.$clock['set'].';'.PHP_EOL;
37
38 \\ Begin module-specific code
39 if($clock['set']==0)
40 {
41     $harness.=' wire CLK_SINK;'.PHP_EOL.' assign CLK_SINK=CLK;'.
        ↪ PHP_EOL;
42     $harness.=" output wire[31:0] CLK_COUNT=32'd0;".PHP_EOL;
43 }
44 else
45     $harness.=" output wire[31:0] CLK_COUNT=32'd".$clock['count'].';'.
        ↪ PHP_EOL;
46 if($assigned['SW']==0)
47     $harness.=' wire[9:0]SW_SINK;'.PHP_EOL.' assign SW_SINK=SW;'.
        ↪ PHP_EOL;
48 if($assigned['KEY']==0)
49     $harness.=' wire[3:0]KEY_SINK;'.PHP_EOL.' assign KEY_SINK=KEY;'.
        ↪ PHP_EOL;
50 if($assigned['LEDR']==0)
51     $harness.=" assign LEDR=10'b0;".PHP_EOL;
52 $harness.=' main u0(' .PHP_EOL;
53 $i=0;
54 $len=count($attach);
55 foreach ($attach as $sig)
56 {
57     $harness.='     '.$sig[1].(' '.$sig[0].')';
58     if($i!=$len-1)
59         $harness.=','.PHP_EOL;
60     else $harness.=')';.PHP_EOL;
61     $i++;
62 }
63 $harness.='endmodule'.PHP_EOL;
64 $harness.='// verilator lint_on UNUSED'.PHP_EOL;
65 $vf=fopen('harness.v','w');
66 fwrite($vf,$harness);
67 fclose($vf);
68 }
69 ?>

```

Listing 4.7: The Generated Verilog Harness

```

1 //verilator lint_off UNUSED
2 module harness(CLK, CLK_EN, SW, KEY, LEDR, CLK_COUNT);
3     input wire CLK;
4     input wire[9:0] SW;
5     input wire[3:0] KEY;

```

```

6   output wire CLK_EN;
7   output wire[9:0] LEDR;
8   assign CLK_EN=1;
9   output wire[31:0] CLK_COUNT=32'd0;
10  main u0(
11      .SW(SW),
12      .LEDR(LEDR),
13      .KEY(KEY),
14      .CLK(CLK));
15  endmodule
16  // verilator lint_on UNUSED

```

Avalon Interface

Altera's IP modules interact using a common bus dubbed Avalon. This embedded hardware interconnect includes multiple interface protocols, including MM, streaming, and interrupt-based, MM being the most prevalent in HDL courses. Ground-level knowledge of this protocol is valuable to any student interested in architecting SoCs. The means to emulate the Avalon MM interface has therefore been included in the DEVBOX prototype.

In order to emulate Avalon's MM interface, two components are necessary. The first component is the MM *Master* port (Figure 4.12). This is the hardware that issues `read` and `write` instructions to attached devices. These peripherals, in turn, have an MM *Slave* port. Figure 4.13 is the block diagram for a simple MM slave. In order for these two ports to communicate, they must adhere to a common protocol. Various signals must be toggled in a pre-determined pattern in order to achieve a successful data transfer. This handshaking process can be represented by the waveform diagram in Figure 4.14. An excerpt from Altera's documentation [16] outlines the details of this waveform as follows:

1. *address, byteenable, and read are asserted after the rising edge of clk. The slave asserts waitrequest, stalling the transfer.*
2. *waitrequest is sampled. Because waitrequest is asserted, the cycle becomes a wait-state. address, read, write, and byteenable remain constant.*
3. *The slave deasserts waitrequest after the rising edge of clk.*
4. *readdata, response and deasserted waitrequest are sampled, completing the transfer.*

Figure 4.12: AvalonMM Master Block Diagram

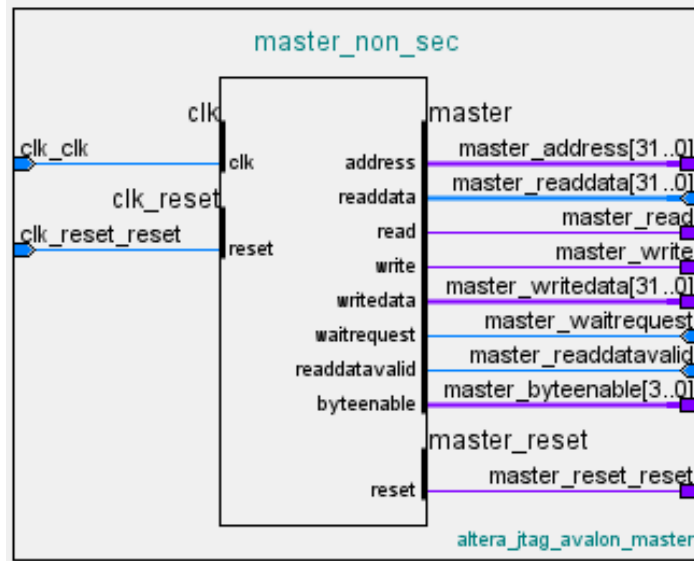
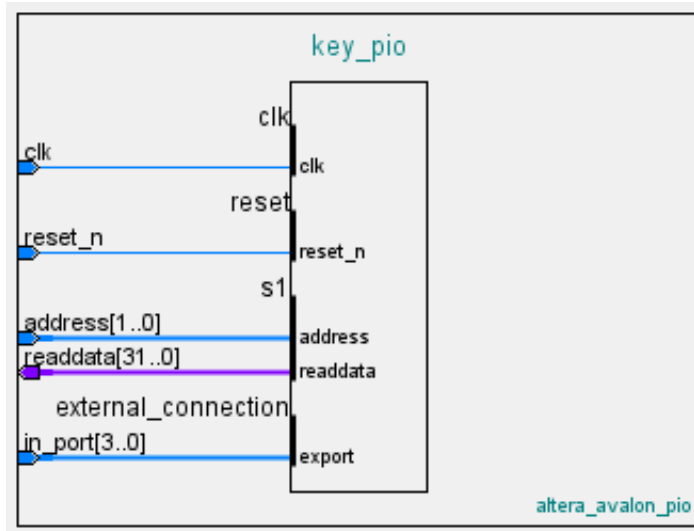


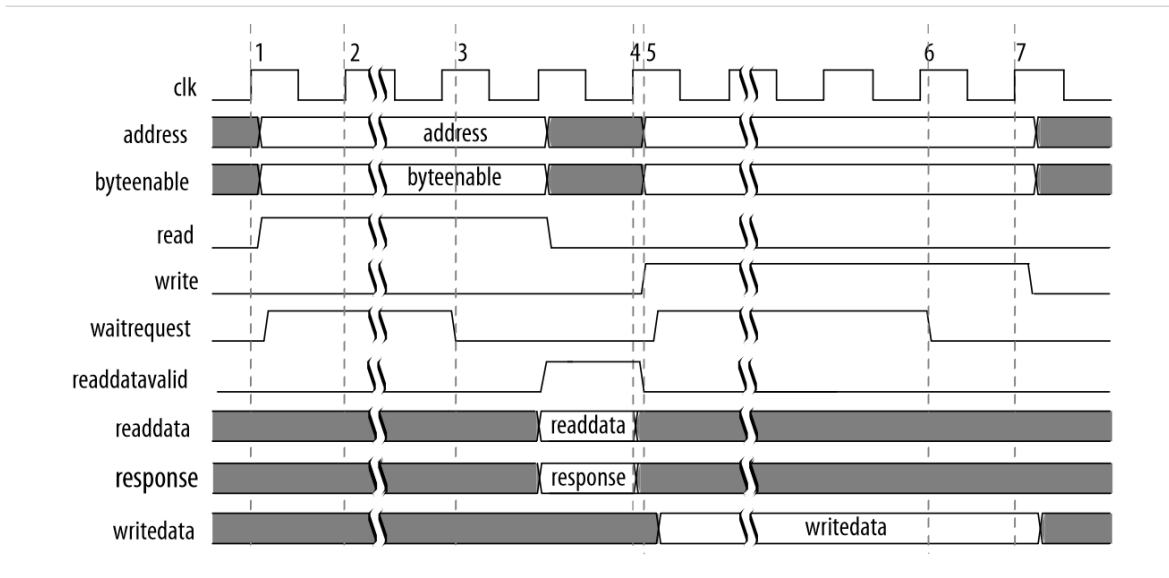
Figure 4.13: Avalon MM Slave Block Diagram



5. *address, writedata, byteenable, and write signals are asserted after the rising edge of `clk`. The slave asserts `waitrequest` stalling the transfer.*
6. *The slave deasserts `waitrequest` after the rising edge of `clk`.*
7. *The slave captures write data ending the transfer.*

The DEVBOX prototype replicates this behaviour using Verilator. The MM Master is approximated in Verilog, providing a module stub, much like the harness module in Figure 4.7, but containing

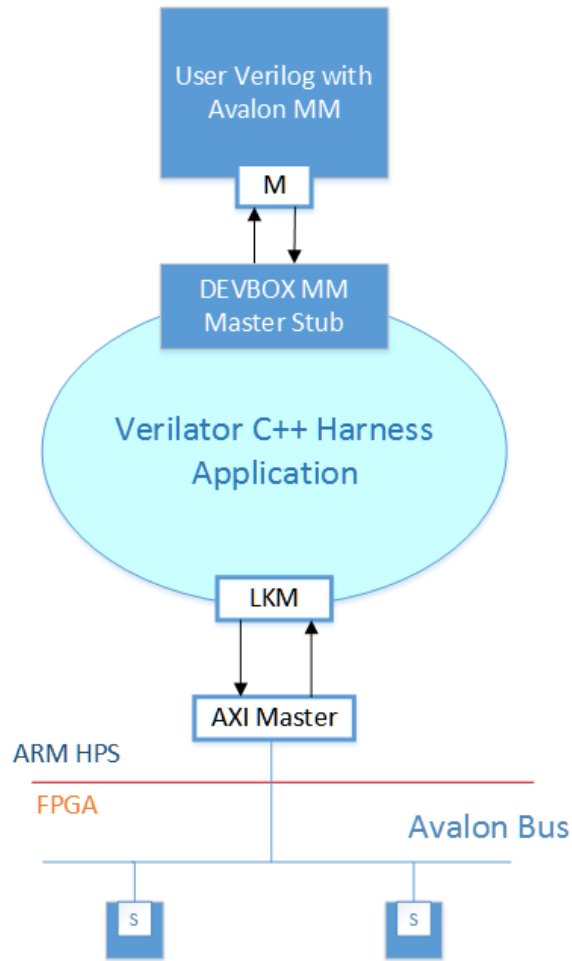
Figure 4.14: Avalon MM read/write Waveform



some finite state machines (FSMs) that manage the signal sequencing for read/write operations.

Listings 4.8 and 4.9 are snippets from the Avalon MM Master interface used by the platform to execute the correct read and write sequences, as visualized in Figure 4.14. Internal registers, such as `cont_write` and `start_read` are used to monitor and control the transition sequence of the FSM. The user's module provides `address`, `byteenable`, and `writedata` asynchronously to the MM Master stub and asserts the `start_read` or `start_write` signals to initiate these sequences. The harness application then emulates the behaviour of the Avalon bus, manipulating the slave-driven signals in Figure 4.14 while transmitting data to and from the addressed MM slave through the LKM. Figure 4.15 shows how the different elements in the DEVBOX Verilog emulation suite interact to simulate an Avalon-MM Master.

Figure 4.15: Emulating Avalon-MM Master



Listing 4.8: The Avalon Write FSM

```
1 // Write Sequence
2 always @ (posedge clk)
3 begin
4     if(!r_read && start_write)
5     begin
6         write<=1;
7         rw_state<=2'b01;
8     end
9     else if(write_state==2'b01)
10    begin
11        write<=1;
12        if(!waitrequest)
13            rw_state<=2'b10;
14    end
15    else if(write_state==2'b10)
16    begin
```

```

17         write<=1;
18         rw_state<=2'b00;
19     end
20     else
21     begin
22         write<=0;
23     end
24 end

```

Listing 4.9: The Avalon Read FSM

```

1     // Read Sequence
2     always @ (posedge clk)
3     begin
4         if(!write && start_read && !readdatavalid)begin
5             r_read<=1;
6             rw_state<=2'b01;
7         end
8         else if(r_read && readdatavalid)
9         begin
10            r_readdata<=readdata;
11            rw_state<=2'b10
12        end
13        else if(rw_state==2'b10)begin
14            r_read<=0;
15            rw_state<=2'b00;
16        end
17    end

```

Likewise, a skeleton project for developing MM slave modules is in place. These stubs allow users to explore the more intricate shared communication channels computer systems use to manage a large number of peripheral devices.

4.7 Summary

The many components described herein have been developed and/or deployed in order to demonstrate the ability of an embedded platform, such as DEVBOX, in delivering a Verilog development education tool. The specifications of the DE-1 SoC board align well with the envisioned hardware while the ARM Linux OS hosts the necessary tools, such as the HTTP daemon and compilation tools. The mock-up web interface successfully demonstrates the ability to develop and test Verilog HDL, as well as C and Arduino, remotely and observe the results on the board's I/O and on a video display. The device drivers that have been developed for the DEVBOX prototype safely expose FPGA hardware

to the web application's back-end software, allowing for more interactive student-developed applications. The inclusion of Avalon stubs and software allow for progression into the more complex aspects of hardware development.

Chapter 5

Testing Procedures

The viability of the DEVBOX platform for instructing pre-university logic design students hinges not only on its ability to present tutorial material in an interactive and intuitive manner, but also on its HDL emulation capabilities. In this chapter, various tests are described that will highlight the current prototype's strengths and limitations in performance, scalability, and ability to emulate university lab assignments in Verilog.

5.1 Verilog Emulation

In order to quantify DEVBOX's viability as a Verilog education platform, it is necessary to identify a point of comparison and unit of measure. The following tests compare DEVBOX's compilation and emulation capabilities to those of a conventional desktop PC.

In section 4.6.3, the use of Verilator as a Verilog emulator is described. It was selected in part because it proclaims itself the fastest free HDL emulator. This is important as it will be operating on a small ARM processor with lower performance than contemporary desktop-grade processors. In order for DEVBOX to operate as a Verilog development tool, it must be capable of parsing, compiling, and running the HDL within an acceptable time frame. It is assumed, when measuring Verilator's performance on DEVBOX's ARM Cortex-A9 dual-core processor that its lower clock frequency will have a considerable impact. ARM's Cortex-A9 architecture allows for variable clock speeds, reducing power consumption during periods of minimal workload. A stress test determined that, during periods

Component	Value
Make & Model	Lenovo ThinkServer
Processor	Intel Xeon E3-1225 V2
Clock Speed	3200 MHz
# Cores	4
Cache Size	8.0 MB
RAM	4.0 GB DDR3-1333
OS	Ubuntu 12.04.5 LTS
Kernel Version	3.2.0-76

Table 5.1: Control PC Specifications

of heavy workload, the Cyclone-V SoC’s Cortex-A9 processor’s maximum clock speed is 800 MHz. The Xeon’s clock speed is 3.20 GHz, 4× faster than ARM’s.

When the operating system reports the execution time of an application, it returns three values:

Real: Start-to-finish clock time

User: CPU time spent in user space

Sys: CPU time spent in kernel space

Although CPU time gives a good indication of processing efficiency, the measure of real elapsed time is the variable that most affects the user.

A baseline for Verilog emulation tasks is acquired by executing them on an x86/64 desktop system with specifications described in Table 5.1.

5.1.1 Compilation

Verilog must be parsed and converted into a C++ class using Verilator’s parser and compiled in conjunction with a top-level harness function into a “Verilated” binary. Depending on the complexity of the design, this can be the most time-consuming step in the emulation process. Benchmarking this process involves comparing the compile time for both simple and complex projects, using Verilator, on both the DE-1 SoC and the PC described above. These results are also compared against Quartus II’s single-threaded compile time for the same project on the same PC.

The reference designs used to compare Verilator’s performance between processors are an open-source Motorola M68K processor, a simple counter and ten copies of the original counter combined

Verilog Project	Lines of HDL
M68K	6709
counter	25
10 counters	438

Table 5.2: Test Verilog Design Complexity

within a top-level module controlling the counters' behaviour. As a measure of complexity, the lines of Verilog code within each design are listed in Table 5.2.

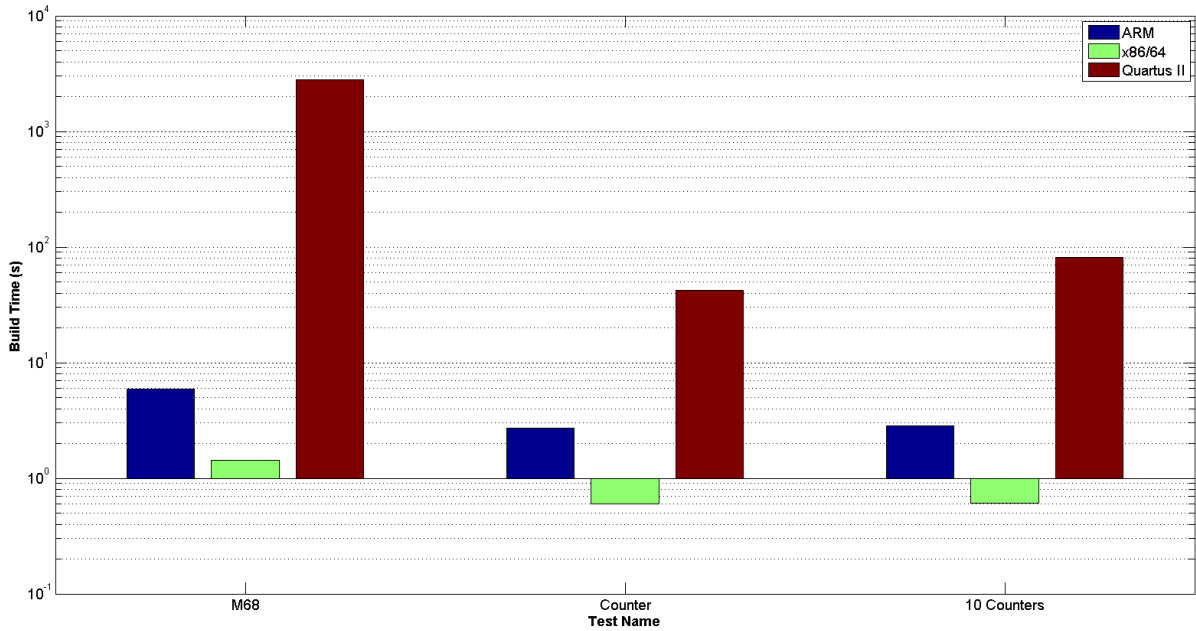
Verilator makes use of thread-level parallelism in its build process by taking advantage of GNU Make's `-jobs` parameter. This feature concurrently executes a specified number of independent jobs in a `Makefile` whose prerequisites have been met. In the case of Verilator, at least six different source files are generated, compiled into objects, and then linked into an executable binary. Much of this work can be completed concurrently. In this way Verilator's compilation stage frequently consumes considerably more CPU time than real time. Ideally, acceleration will scale with the number of cores available.

For this experiment, the designs are compiled multiple times on the ARM and x86/64 cores and then synthesized in Quartus II and real execution time is then recorded. The geometric mean of these results is then reported. The goal is to determine if the time elapsed in compiling with Verilator on the ARM processor is tolerable in comparison.

Results

The process of producing SRAM Object Files (sofs), bitstream files used to encode hardware descriptions on FPGAs, in Quartus II, Altera's IDE, is lengthy. It consists of multiple complex stages designed to optimize the performance, timing, and placement of the hardware on FPGAs and Complex Programmable Logic Devices (CPLDs). When emulating hardware using an application such as Verilator, the necessity for such in-depth analysis is eliminated. It suffices to approximate the Register-Transfer Level (RTL) behaviour of the HDL code. Figure 5.1 shows that, for the reference designs used for these tests, Quartus II's processing time is considerably greater than Verilator's on the reference PC and on DEVBOX itself. In order to get a better idea of how DEVBOX's ARM processor compares to conventional x86/64 processors found in modern PC and server computers, the Quartus

Figure 5.1: Verilator Build Times



results should be disregarded in further studies.

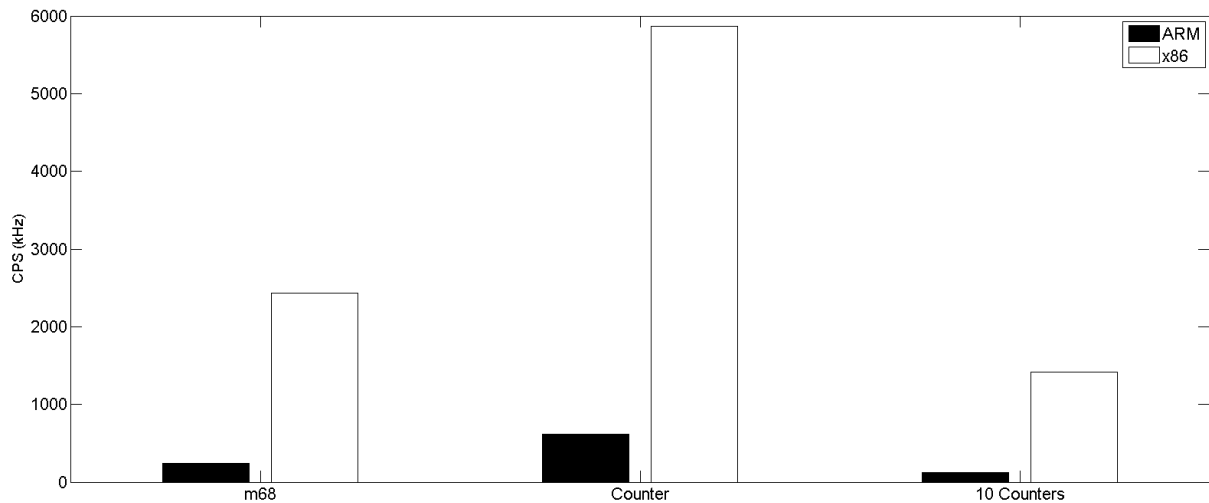
The desktop processor is able to compile Verilog 20 times faster with Verilator than the 2-core ARM processor. However, it is important to note that the ARM processor under test contains only two cores, compared to the 4-core Xeon processor in the control system. Therefore, the mean per-core speedup is only 10 \times .

5.1.2 Execution

Once the Verilog is compiled into an executable binary, it is run on the host system and execution time is measured. The harness application runs the design for a fixed number of emulated clock cycles. Then, using the execution time, the emulation's Cycles per Second (CPS), or effective clock frequency, is calculated. Comparing this metric for both the ARM and x86/64 processors provides a good representation of the Cortex A9's effectiveness at emulating Verilator-compiled hardware modules.

The M68K project contains an application written in assembly and a data file as an input vector for its simulation. The counter modules are also provided with input to ensure that they are active

Figure 5.2: Verilator Execution Times



throughout the simulation.

The executable emulation applications compiled in the previous experiment are run multiple times as in the compilation test. As before, the geometric mean of the resultant CPS are used to compare systems.

Results

When comparing the execution speeds of Verilator-generated, or “Verilated”, emulations between processor architectures, it is important to note that the resulting applications are not multi-threaded. This means that the entire emulation is processed on a single core. Therefore, any disparity between the results will be due to the difference in clock speeds and processing efficiency.

The results show that the reference PC was almost $10\times$ as effective as the ARM system at emulating Verilated hardware. This correlates well with the compilation findings in Figure 5.1. Despite the slowdown in simulation speed, the ARM Cortex-A9 is still capable of delivering roughly 240 kiloHertz (kHz) to the complex Motorola M68K design.

5.1.3 Interpretation

The results of the tests described above are an indication of Verilator’s capabilities as a Verilog emulator on DEVBOX’s ARM Cortex-A9. ARM’s architecture is focused on minimizing power consumption. With a lower clock speed, power-optimized caches, and a variety of other optimizations, it sacrifices raw compute power for energy savings, making it ideal for portable embedded systems and application-specific devices. As such, it is no surprise that powerhouse processors, such as the Intel Xeon processor in the control PC used in the benchmarks above, would be notably faster at executing the necessary tasks for Verilog emulation with Verilator.

The inclusion of a four-core Cortex-A9 to the system’s architecture would likely reduce Verilator’s compilation time by up to 50%, performing more “jobs” concurrently but would have no impact on emulation.

Emulation at 240 kHz is respectable for the purpose of debugging and learning to code and students will not notice much lag between input and output as their code is executed.

That said, more effective emulation models, such as the ZUMA embedded FPGA (eFPGA), may soon be available for ARM processors. This will potentially improve both the speed of compilation and emulation, thus removing the discontinuity observed in these test results and taking advantage of the reconfigurable hardware. Such a tool will be a valuable addition to future iterations of the DEVBOX toolset.

5.2 Verilog Scalability

Another indication of DEVBOX’s effectiveness at compiling and emulating Verilog natively is its efficiency as designs grow in complexity or size. Using the “10 Counters” module from the previous tests, an experiment was developed to measure the processing time for both compilation and emulation as the number of counters increases. The test is run on the DEVBOX prototype’s ARM processor, sequentially building and executing counter designs with an increasing number of counter instantiations. The emulation is again run for a fixed number of clock cycles and the real and execution times are recorded and compared.

5.2.1 Compilation

As described in Section 5.1.1, Verilator parses Verilog and converts it into C++, and uses GCC to build an executable binary. In order to dynamically modify the size of the “Counters” design, changes were made to the Verilog module to allow for size to be defined by a precompile directive. By defining `COUNT=$COUNT` at the command line, the top-level Verilog module uses the given value as a parameter in a `generate` loop to instantiate `$COUNT` copies of the counter module. With this change in place, a command line script easily iterates over increasingly large designs and records compilation time.

Verilator makes use of thread-level parallelism in its build process by taking advantage of GNU Make’s `-jobs` parameter by default. In this execution mode, all build stages, whose prerequisite objects are ready, are launched simultaneously. It is therefore expected that real execution times should be notably shorter than the CPU times.

Results

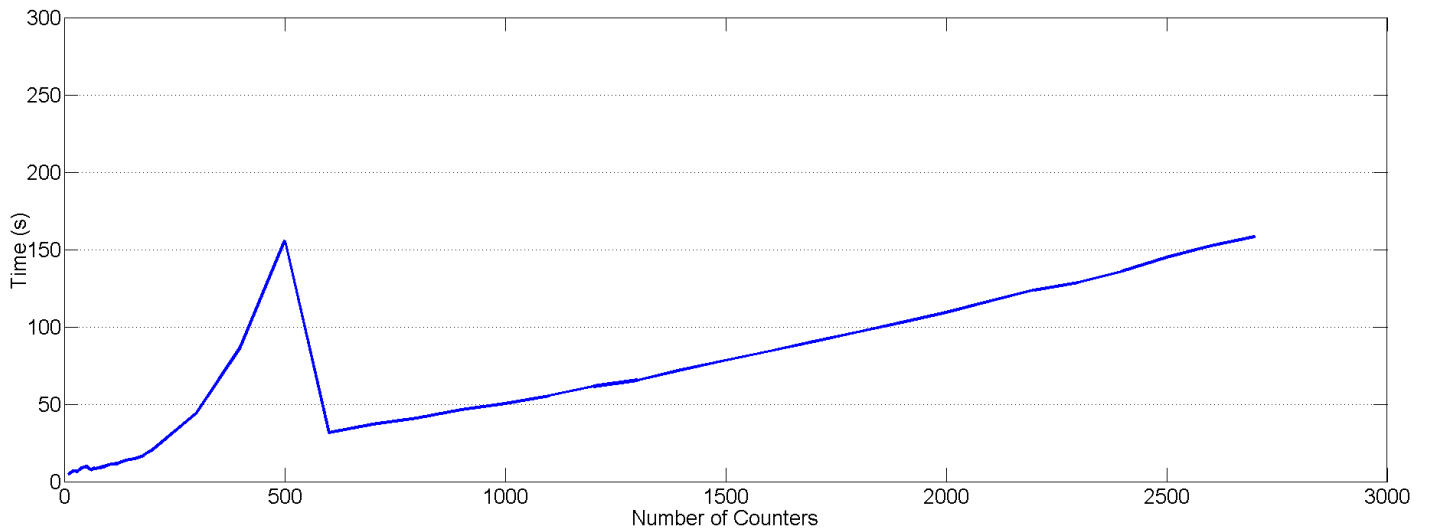
The amount of time elapsed during Verilator compilation on an ARM core is not trivial, on the scale of minutes. Compiling a single counter module takes only a few seconds, but when the size of the model is increased, the compilation runtime grows. As seen in Figure 5.3, CPU time exceeds real processing time, suggesting execution of this phase appears to be computation-bound and parallelized. More cores on the processor will potentially have a positive effect on compile times. More advanced ARM processors, such as the quad-core Cortex-A15 [7], have increased multiprocessing functionality over the Cortex-A9 and may prove more efficient at compiling larger models.

The results of the compilation scalability benchmark exposes an anomaly in processing time as the number of counter modules exceeds 500. After this point, compilation time drops significantly and the slope of the curve is much shallower than for smaller models. We are unable to explain this behaviour.

5.2.2 Execution

The DEVBOX prototype’s scalability with respect to project size is also important. Verilog emulation on the DEVBOX platform is meant, in general, for experimentation and debugging purposes, provid-

Figure 5.3: Scalability of Compilation for ARM Verilator



ing a testbed for new HDL students. The performance of this system in emulating RTL behaviours for small designs is a determining usability factor. As project complexity increases over the course of a student’s education, the emulation speed must be tolerable.

To this end, the C++ harness from the compilation scalability test is modified to manipulate the counters as they increase in number. the script described in Section 5.2.1 then runs the generated executable after each subsequent compilation and records its runtime.

Verilator has yet to optimize its RTE for concurrent processing, so real execution time is predicted to be equal to or greater than its CPU time.

Results

Similar to the compilation results there is a similar anomaly in clock speed when the number of counter modules grows. Figure 5.4 shows the variation in CPS as the module increases in size. There is a lack of consistency in results until the number of counters exceeds 200, at which point the speed declines more linearly. Figure 5.5 compares the real execution time against processor time. CPU time displays a near identical anomaly to the compilation results, dropping at 500 counters. Again, we are unable to ascertain the cause.

Figure 5.4: Scalability of Execution for ARM Verilator

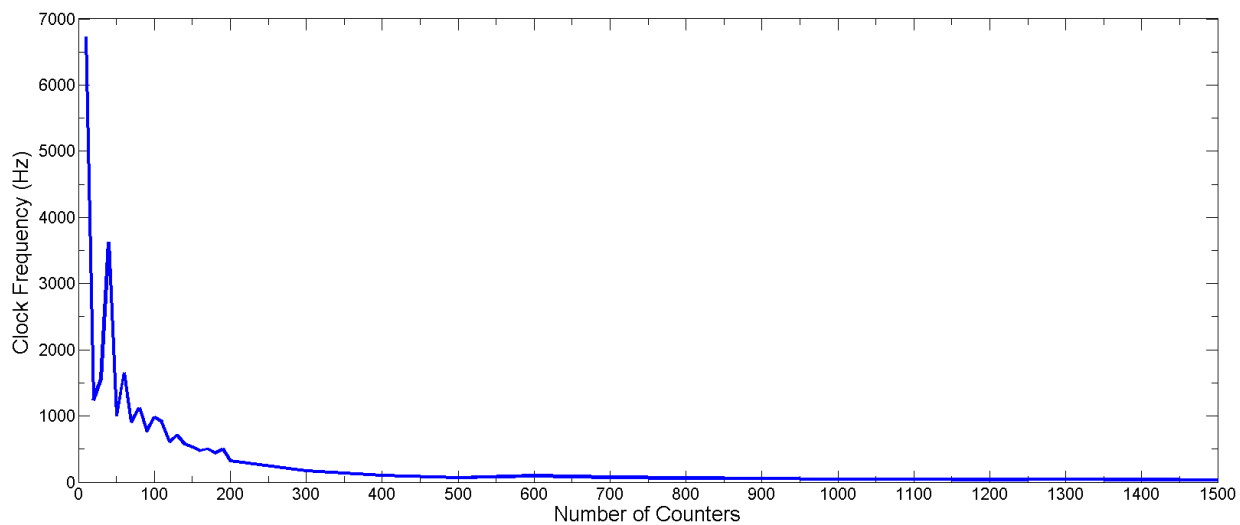
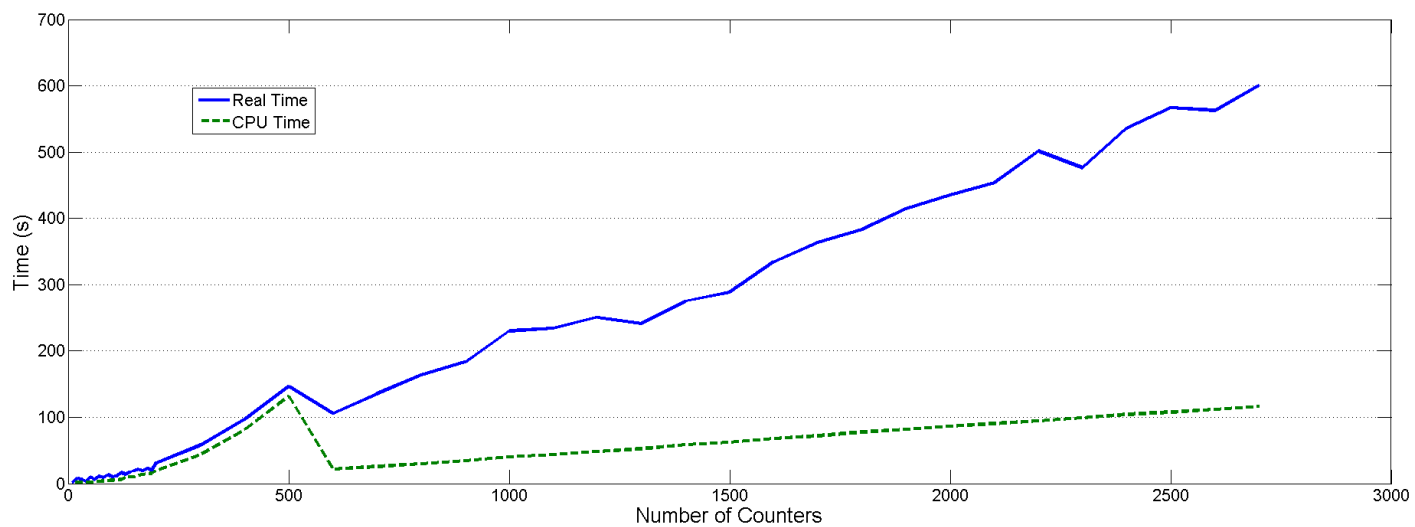


Figure 5.5: Real vs. CPU Execution Times



5.2.3 Interpretation

Analysis of the previous results for compilation suggests that, for small hardware designs, Verilator may not scale well. However, after a certain threshold, larger projects compile faster and scale much more linearly. Similarly, CPU time for execution shows the same problem, but is not as apparent with real execution time.

5.3 Course Material

An important benchmark for the DEVBOX prototype is its ability to implement the projects assigned to University-level ECE students. The lab assignments from UBC's digital logic design course, EECE353/CPEN311, will serve to demonstrate the platform's ability to emulate Verilog for students of HDLs and logic design. These lab assignments range in complexity from simple adders to video games. They use many of the on-board I/Os devices at a bare-wire level and the resultant hardware is designed to be interactive in nature. These assignments are ideal for exploring the software DEVBOX uses to emulate Verilog for the purpose of self-directed HDL education. They are written with Altera's DE-2 board in mind, but can, for the most part, be adapted to the DEVBOX prototype's board.

In order to assess the platform's viability for providing instructional materials to students, we completed the lab assignments from the above course in Verilog using DEVBOX, and tested them according to the specifications outlined in the assignment documentation. Noticeable lag or transient output errors were also catalogued. There are five labs, increasing in complexity throughout the curriculum, and each is completed using only the provided instructions and generic on-line resources. Appendix D contains the lab assignments from 2013 and 2014 chosen for this test. Assignments involving Quartus II or Qsys design software are omitted as these tools are not available for ARM Linux. Additionally, there are some assignments that include the use of hardware that is not available, such as the LCD display found on the DE-2 board.

Brief descriptions of each attempted lab's goals are listed below.

2013-Lab1: This assignment requires the development of a 7-segment decoder combinational circuit that displays one of digits 0 through 7 on a 7-segment display based on a 3-bit input from the board's switches.

2013-Lab2: The first portion of this assignment requires the use of an on-board LCD display and is omitted. The second half involves developing a craps dice game using an FSM and modular design hierarchy.

2014-Lab2: The first half of this assignment is the same LCD display task as in the previous year. For the second half, a simple VGA pixel writer is designed to interact with an open source VGA adapter module.

2014-Lab3: This assignment has the student develop a blackjack game using buttons and switches as input and 7-segment displays and LEDs as output.

2014-Lab4: For this assignment, the challenge is to develop hardware versions of line-drawing algorithms for the adapter module from assignment 2.

A short list of modifications had to be made to support this selection of tasks on the DEVBOX prototype. First, a much larger region of physical memory addresses had to be exposed in order to access the entire array of MM I/O available on the FPGA. The API for this driver was simplified to read and write functions of variable bit-width, similar to the Avalon API. Additionally, a new harness application was required to accommodate the new interface, which necessitated a new top-level Verilog module. With these elements in place, the assignments were completed, following the instructions in materials presented to Logic Design students.

5.3.1 Results

2013-Lab1:

For this assignment, the character display can be easily written with a single asynchronous `case` statement. The student must demonstrate a circuit where all 7-segment, or HEX, displays present the decimal value of a 3-bit integer selected by `SW[2:0]`. This circuit is then modified to display four different numbers on four individual HEX displays by instantiating multiple instances of a single module. As a bonus task, the circuit is modified again to display the letters A through H instead of digits. Table 5.3 logs whether or not the task is demonstrated successfully as well as any detectable

Task	Result	Emulation Anomalies
Task 5: All HEX display same digit	Successful	notable Δt between HEX transitions
Task 6: Independent HEX displays	Successful	Only 3 different values displayed due to lack of switches
Task 7: Modified to display alpha characters	Successful	Same as Task 6

Table 5.3: Lab 1 Assignment Results

anomalies or variations in behaviour observed during emulation. This module's compile time was 19.45s.

2013-Lab2:

Due to the lack of an LCD display on the DE1-SoC board, the first task of this assignment is omitted from these results. For the second half, multiple modules, both complete and to be completed by the student, are provided in the course material. These modules are written in VHDL and were rewritten in Verilog for the purpose of this test¹.

The goal of this task is to develop an FSM in HDL that adheres to the behavioural description, listed in Appendix D.1, in the lab assignment.

In Table 5.4, the test results indicate that DEVBOX's Verilog emulation was able to demonstrate all of the requirements for this lab assignment but exhibited a detectable transition delay on the physical I/O. Also, a bug in the application caused an error that propagated through the last two states of the machine. This is possibly due to a logic error in the Verilog code itself or in the C++ harness. The craps module compiled in 18.55s.

2014-Lab2:

The first section of this assignment is identical to that of the previous year's and has been omitted for the same reason. The goal of the second half of this assignment is to interface with a custom VGA adapter to manipulate the pixels of the attached screen. As before, many complete and partial modules are included in VHDL and most have been rewritten in Verilog for the purpose of this test. One

¹In HDL IDEs, the use of Verilog and VHDL are generally interchangeable, which means modules written in VHDL are directly portable to Verilog projects. The same cannot be said of Verilator. It is specific to Verilog and will not translate VHDL modules into C++.

State	Result	Emulation Anomalies
1. Init: -- dice values, LEDR[0] on.	Success	None
2. First Roll: 1-6 on HEX1 and HEX0. Total on HEX3 and HEX2.	Success	Delay between 7-segment transition and LEDR transition
2a. Win: LEDG on and LEDR[0] off	Success	LEDR[3:1] used due to lack of LEDG
2b. Lose: Light up LEDR[17:1] and LEDR[0] off	Success	LEDR[9:7] used as per anomaly 1b.
2c. Roll Again: Total is value to match and copied to HEX5 and HEX4	Success	Delay on HEX5, HEX4 transition
3. Reroll: New roll values on HEX1, HEX0. New total on HEX3, HEX2	Success	None
3b. Match Win: Same as 2a.	Logic Error	LED state fails to update until next reroll
4. Game Over: Maintain state of outputs. wait for reset.	Success	Due to logic error in 3b. The wrong dice value and total are displayed in bottom four HEXs in event of match win.

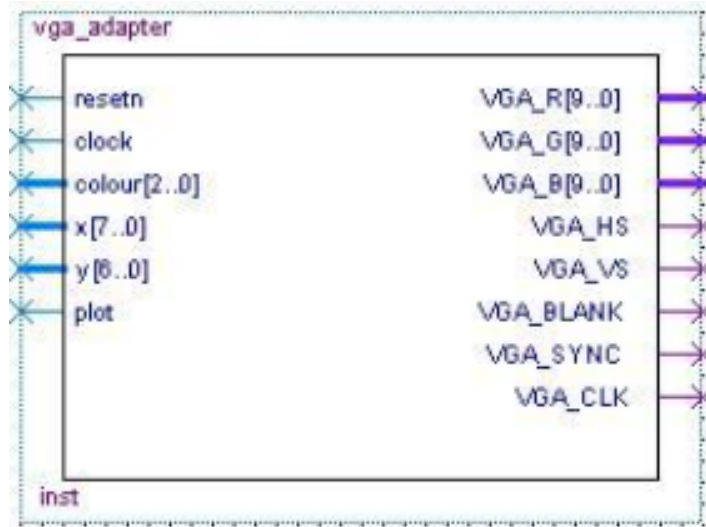
Table 5.4: Lab 2 (2013) Assignment Results

hardware module, however, contains a relatively high degree of complexity and was not translated. This module is the VGA adapter mentioned above. Because it is immutable in the context of this assignment and interfaces directly with the VGA hardware, it was possible to avoid its use for this test.

Figure 5.6 is a black-box representation of the adapter interface. The signals on the left-hand side are driven by the student's hardware design. The signals on the right drive the VGA DAC directly. In order to bypass this module, its behaviour was built into the C++ harness Verilator uses to drive the emulation. The port list for this assignment's top-level entity contained the left-hand signals as outputs rather than those on the right-hand side, as would be expected in the original assignment. The harness program then uses this output with the `fbDraw` library described in section 4.5 to emulate the adapter's behaviour.

There are two tasks that are to be demonstrated in this assignment. The first is to fill the screen in with a single colour. The second is an optional task where a 4×4 box is drawn at a location specified by the switches. Although this bonus circuit was completed and tested successfully, its complexity is trivial and has been omitted from this test. The screen resolution used is 160×120 pixels.

Figure 5.6: VGA Adapter Core as a Black Box



Upon the completion of the circuit for the main task, the following observations were made:

1. The pixels were drawn in the correct order.
2. The pixels were set to the correct colour.
3. The entire screen was filled by the circuit.
4. Total time to fill screen is approximately 40 minutes.

The first three points satisfy the assignment's requirements in that the circuit successfully fills the screen with a single colour. The final point, however, renders the emulation of hardware-driven graphics unrealistic.

Frame rates, measured in Hertz (Hz) or $\frac{\text{frames}}{\text{second}}$, indicate the time taken to redraw the entire screen and should be 24 Hz or greater for smooth video. If the Verilated circuit requires 40 minutes to fill the entire screen, the emulator's effective frame rate is 4.167×10^{-4} Hz. This means it is only capable of drawing eight pixels every second. This module required 17.94s to compile.

2014-Lab3:

The goal of this lab assignment is to develop a non-trivial FSM to drive a Blackjack game. The required sequence of steps to successfully complete this circuit are as follows:

1. **Start.** Give player a card.
2. Give dealer a card.
3. Give player a second card. If player stands, go to **DealersTurn**.
4. Give player a third card. If player stands or goes bust, go to **DealersTurn**.
5. Give player a fourth card.
6. **DealersTurn.** Give dealer a second card. If dealer stands, go to **Winner**.
7. Give dealer a third card. If dealer stands or goes bust, go to **Winner**.
8. Give dealer a fourth card.
9. **Winner.** Decide winner:
 - If both go bust, there is no winner. Go to **EndGame**.
 - If dealer \geq player, or player is bust, go to **DealerWins**
 - If dealer $<$ player, or dealer is bust, go to **PlayerWins**.
10. **DealerWins.** Turn on LEDR [17 : 0]. Go to **EndGame**.
11. **PlayerWins.** Turn on LEDG [7 : 0]. Go to **EndGame**.
12. **EndGame.** Wait forever.

The HDL developed for this test displayed no observable anomalies that could be attributable to Verilator's effectiveness at emulating Verilog on the DE1-SoC's ARM processor. This module required the most time to compile at 23.86s.

2014-Lab4:

This assignment makes use of the same open-source VGA adapter used in **2014-Lab2** to implement line-drawing algorithms. Due to the slow performance experienced in the aforementioned lab, debugging the hardware design reached a near standstill. Development was abandoned before the requirements of the assignment had been met.

5.3.2 Interpretation

The results from the course material tests indicate that the platform is capable of emulating simple “Verilated” designs with an acceptable amount of observable anomalies in the form of delays between I/O transitions. However it is ineffective at emulating more complex designs such as VGA graphics.

This observation compels the use of a more efficient method of Verilog hardware emulation for use in DEVBOX. In order to deliver an effective training experience, the platform must at least perform the required tasks of an early-university logic design course.

Chapter 6

Future Work

The DEVBOX platform is still in the early stages of development and there are many avenues for progress to be explored. In this chapter some of the new components and improvements will be described and discussed.

6.1 Web Interface

The current browser-based GUI for DEVBOX is a simple mock-up of the intended suite. Many improvements and additions can be made to improve the intuitiveness and effectiveness of the interface, as well as its look and feel.

6.1.1 Editor and Debugging

The prototype's editor screen is a bare-bones example of the basic tools required to build and run code. It lacks support for many of the debugging and analysis tools commonly available in most IDEs. Breakpoint integration, auto-completion, and context-sensitive menus are all editing tools that can provide a much more productive learning environment. Particularly in testing Verilog, bugs can be incredibly difficult to find if you are unable to step through execution or view the waveform of a simulation. In the next iteration of the web application these are features that should be integrated into the *Editor* page.

6.1.2 Tutorials

The current prototype of the *Tutorials* page includes a text area for displaying documentation and a scaled version of the editor. The text and code are combined into a single file to be parsed together as the tutorial progresses from page to page, dynamically updating the program code to coincide with the page's challenge. The parser, dubbed *TutoriML*, currently performs this task on a purely cosmetic level. In the future this method of combining text with code for the purpose of dynamic tutorial presentation should be refined to a seamless mechanism to allow students to enter their own code into the sample program without having it overwritten when moving to the next stage.

A subjective test can also be performed to compare the current presentation style to an alternate configuration that does not require turning pages, as well as to the current browser-and-IDE two-window method used by students. This would ideally be conducted across multiple high school campuses with a diverse cross-section of secondary students.

There are only a handful of tutorials available on the prototype, and they are mostly intended only demonstrate the concepts of interactivity provided by TutoriML. A large body of well-written tutorials will be required to render the *Tutorials* page effective. These could be adapted from free online resources or written specifically for the DEVBOX environment, focusing on the available hardware I/Os. Quality Verilog training material outside of university-level courses is uncommon. Developing quality introductory tutorials for logic design, computer architecture, and HDLs should be a priority in future iterations.

6.1.3 Templates

Another addition to be made to the web interface is the inclusion of a *Templates* page. This page will host a library of bare-bones source files outlining the procedures for certain types of programs. These starting points will help new students jump ahead to the meat of the programming experience where tangible results come often.

6.2 Verilog Simulation

6.2.1 ZUMA eFPGA

As discussed in Chapter 5, Verilator’s emulation performance on the DEVBOX prototype is very slow for non-trivial solutions. While it is possible that other software-based emulators may be faster on ARM processors than Verilator, another, potentially much faster, option may be available in the near future. The ZUMA eFPGA is able to provide hardware accelerated HDL emulation in a similar fashion to synthesized hardware without the dependence on commercial tools such as Quartus II. An open-source bitstream generator, called “Verilog to RTL (VTR)”, synthesizes the Verilog into a file that runs on the ZUMA architecture, like an FPGA within an FPGA, providing a considerable speedup over Verilator emulation. The ZUMA toolchain and eFPGA will be integrated into the DEVBOX platform as it becomes available for ARM processors.

6.2.2 Debugging

Future iterations of the HDL emulation tools should include detailed debugging modes. These will allow students to observe the progression of a hardware design, monitor signal values and step through lines of Verilog while the hardware I/Os continue to reflect the system state as they would with synthesized hardware. Specifically, the following need to be implemented:

- Breakpoints
- Signal taps
- Clock- and line-stepping

6.3 Community Websites

As the DEVBOX platform continues to evolve, the community will become increasingly important. Enticing participants to engage in the DEVBOX ecosystem will enrich the learning experience for students and motivate innovation among the developers. An easy to understand and welcoming user interface with fluid and dynamic interaction models can be developed to enable source code sharing,

distribution of documentation, and general social interaction such as chat and multi-user editing of the same code.

6.4 Device Prototype

The current DEVBOX prototype resides on a Terasic DE1-SoC development board which does not contain all of the necessary hardware elements that the concept requires. Future design and development of the final device will include these missing elements, such as WiFi, Arduino Shield General Purpose I/Os (GPIOs), and HDMI output.

Chapter 7

Conclusion

DEVBOX is a versatile development education platform intended to accommodate students and enthusiasts with a broad range of experience in multiple programming languages. The viability of this platform as a tool for introducing Verilog to students early in their academic career is explored in this thesis. By providing a platform for a hands-on independent learning tool for HDL and logic design with no dependence on lengthy downloads, software installation or driver conflicts, DEVBOX aims to eliminate the barrier-to-entry and intimidation factor faced by prospective students of computer architecture, SoC development, and logic design.

A large body of work has been done to facilitate the grade school to early university education of computer programming students by way of VPLs, laptops, and embedded computer systems preloaded with development tools and more intuitive course and tutorial material. In contrast, very little has been done to date toward improving the learning experience in digital logic and HDLs. The subject matter is generally considered to be prohibitively complicated, and left to sophomore engineering courses. Many studies have shown that early introduction to computer programming and related subjects greatly improves success rates among first year students in technology-related fields, and the same might be inferred with regard to Verilog.

This thesis proposes the use of an embedded system with hardware and software tailored toward self-directed programming instruction, with a zero-install, zero-setup configuration, such as the DEVBOX platform described in Chapter 3. Such a platform is capable of delivering the instruction and

development material required for early HDL education. This includes tutorials, compilation and emulation tools, visual feedback and hard I/O integration.

DEVBOX was originally proposed as an education tool for prospective software developers, but has shown potential for delivering the necessary materials for early logic design education. Learning HDL can be greater challenge for many computer science and ECE students than learning software programming languages. Introducing the concepts and syntax involved in hardware description prior to university at a pace determined by the student will have a positive impact on student success rates in early-year logic design courses.

A prototypical model of DEVBOX has been developed, as described in Chapter 4, that is capable of providing the required elements described above for not only Verilog HDL, but also for C and Arduino C programming languages. It natively compiles these three languages and runs the generated programs internally, receiving input from and exporting output to the user over a local network and on-board I/O devices. A browser-based mock-up web application provides a GUI through which users can access the provided resources. The “*Tutorials*” and “*Editor*” pages are both functional and demonstrate the minimal desired behaviour of these interfaces.

By creating a prototype device that does not require the installation of specialized software or the use of a specific operating system, and requires access only to common household resources, such as WiFi and an HDMI-ready display, the development of the DEVBOX prototype is a viable candidate for producing a portable and easy-to-use development education platform. The UI mock-up uses asynchronous JavaScript, PHP, and Node.js to provide a dynamic development environment. Tutorial content is also delivered in a dynamic way, using the custom-developed TutoriML syntax and parser to allow users to test and modify code as they navigate through their sessions.

The DEVBOX prototype simulates Verilog by way of the Verilator HDL simulator and makes use of the device’s hard I/Os in the same way synthesized modules would on an FPGA. A series of tests has unveiled the strengths and shortcomings of the DEVBOX prototype in simulating Verilog and providing an instructional environment for logic design and HDL. These tests have demonstrated that, despite reliably emulating simple Verilog modules in an ARM Linux environment, it lacks the efficiency to perform more complex tasks. For example, if a “Verilated” VGA adapter were to render

a single High Definition (HD) frame, the process would take approximately three days.

More effective emulation methods are currently being researched and ported to the ARM Instruction Set Architectures (ISAs) that may increase the speed and efficiency with which students' designs are executed. These eFPGAs have the potential to drastically increase emulation speed. It applies an open FPGA framework over top of the Cyclone V-SoC's proprietary gate array and compiles Verilog into this framework with open source software.

Based on the current progress and test results described in this document, an embedded SoC system such as the one described for the DEVBOX platform is capable of delivering the tools necessary for early logic design and HDL education. Despite emulation efficiency concerns related to more complex designs, such as multimedia and DSP, the concept devised herein has been successfully prototyped and proven to provide a platform for a zero-install, interactive, and effective HDL education ecosystem.

Bibliography

- [1] Arduino - Home. URL <https://www.arduino.cc/>. [Last accessed 2015-09-22]. → pages 10
- [2] Bart0vds's Profile - Member List - Minecraft Forum. URL <http://www.minecraftforum.net/members/Bart0vds/reputation>. [Last accessed 2015-07-26]. → pages 2
- [3] BeagleBoard.org - community supported open hardware computers for making. URL <http://beagleboard.org/>. [Last accessed 2015-09-22]. → pages 9
- [4] "clang" C Language Family Frontend for LLVM. URL <http://clang.llvm.org/>. [Last accessed 2015-10-13]. → pages 54
- [5] Code::Blocks, . URL <http://www.codeblocks.org/>. [Last accessed 2016-01-22]. → pages 8
- [6] Codeboard the IDE for the classroom, . URL <https://codeboard.io/>. [Last accessed 2015-10-13]. → pages 6, 7
- [7] Cortex-A15 Processor - ARM, . URL <http://www.arm.com/products/processors/cortex-a/cortex-a15.php>. [Last accessed 2015-11-30]. → pages 73
- [8] Eclipse - The Eclipse Foundation open source community website. URL <http://www.eclipse.org/>. [Last accessed 2016-01-22]. → pages 8
- [9] Minecraft. URL <https://minecraft.net/>. [Last accessed 2015-07-26]. → pages 2
- [10] One Laptop per Child. URL <http://one.laptop.org/>. [Last accessed 2015-09-22]. → pages 8
- [11] Raspberry Pi - Teach, Learn, and Make with Raspberry Pi. URL <https://www.raspberrypi.org/>. [Last accessed 2015-09-22]. → pages 9
- [12] Yocto Project | Open Source embedded Linux build system, package metadata and SDK generator. URL <https://www.yoctoproject.org/>. [Last accessed 2016-01-22]. → pages 44
- [13] AMBA Specifications - ARM. URL <http://www.arm.com/products/system-ip/amba-specifications.php>. [Last accessed 2015-09-22]. → pages 37
- [14] Cyclone V - Overview. URL <https://www.altera.com/products/fpga/cyclone-series/cyclone-v/overview.html>. [Last accessed 2015-09-22]. → pages 36

- [15] Node.js. URL <https://nodejs.org/en/>. [Last accessed 2015-05-27]. → pages 45
- [16] Altera Corporation. Avalon Interface Specifications, Mar. 2015. URL https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf. [Last accessed 2015-11-12]. → pages 61
- [17] M. D. L. n. Cifredo-Chacn, n. Quirs-Olozbal, and J. M. Guerrero-Rodrguez. Computer architecture and FPGAs: A learning-by-doing methodology for digital-native students: COMPUTER ARCHITECTURE AND FPGAs. *Computer Applications in Engineering Education*, 23(3):464–470, May 2015. ISSN 10613773. doi:10.1002/cae.21617. URL <http://doi.wiley.com/10.1002/cae.21617>. → pages 4, 15
- [18] M. Conway, S. Audia, T. Burnette, D. Cosgrove, K. Christiansen, R. Deline, J. Durbin, R. Gossweiler, S. Koga, C. Long, and others. Alice: Lessons Learned from Building a 3d System For Novices. 1999. URL <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.43.2242&rep=rep1&type=pdf>. → pages 5
- [19] S. Cooper, W. Dann, and R. Pausch. Alice: a 3-D tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, volume 15, pages 107–116. Consortium for Computing Sciences in Colleges, 2000. URL <http://dl.acm.org/citation.cfm?id=364161>. → pages 5
- [20] Y.-C. Hung. The Effect of Problem-Solving Instruction on Computer Engineering Majors’ Performance in Verilog Programming. *IEEE Transactions on Education*, 51(1):131–137, 2008. ISSN 0018-9359. doi:10.1109/TE.2007.906912. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4448425>. → pages 15
- [21] B. Isong. A Methodology for Teaching Computer Programming: first year students perspective. *International Journal of Modern Education and Computer Science*, 6(9):15–21, Sept. 2014. ISSN 20750161, 2075017X. doi:10.5815/ijmecs.2014.09.03. URL <http://www.mecs-press.org/ijmecs/ijmecs-v6-n9/v6n9-3.html>. → pages 13
- [22] L. Jeniric. Teaching Introductory Programming. *ijacsa*, 5(6):60–69, July 2014. doi:10.14569/IJACSA.2014.050611. → pages 13
- [23] S. K. R. K. and S. Kode. Enhancing the Learning Experience by Addressing the Needs of the Learner Through Customization and Personalization in the Learning by Doing Methodology. pages 274–275. IEEE, July 2010. ISBN 978-1-4244-7144-7. doi:10.1109/ICALT.2010.80. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5571306>. → pages 4
- [24] Z. Keaton. Kano Launches New Computer Kit. *Techcrunch Disrupt 2015*, May 2015. URL https://dl.dropboxusercontent.com/content_link/bEkGhO87y0IH0iH1WgQNfZ3h71tmaQ00Jkk9Vr4eqNZeYGVomPMLXn3GJXwJjqb?dl=1. → pages 10
- [25] M. Klling. The Greenfoot Programming Environment. *ACM Transactions on Computing Education*, 10(4):1–21, Nov. 2010. ISSN 19466226. doi:10.1145/1868358.1868361. URL <http://portal.acm.org/citation.cfm?doid=1868358.1868361>. → pages 6

- [26] T. Koulouri, S. Lauria, and R. D. Macredie. Teaching Introductory Programming: A Quantitative Evaluation of Different Approaches. *ACM Transactions on Computing Education*, 14(4):1–28, Dec. 2014. ISSN 19466226. doi:10.1145/2662412. URL <http://dl.acm.org/citation.cfm?doid=2698235.2662412>. → pages 12
- [27] G. Licea, R. Jurez-Ramrez, C. Gaxiola, L. Aguilar, and L. G. Martnez. Teaching object-oriented programming with AEIOU: OBJECT-ORIENTED PROGRAMMING. *Computer Applications in Engineering Education*, 22(2):309–319, June 2014. ISSN 10613773. doi:10.1002/cae.20556. URL <http://doi.wiley.com/10.1002/cae.20556>. → pages 12
- [28] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick. Scratch: a sneak preview [education]. In *Creating, Connecting and Collaborating through Computing, 2004. Proceedings. Second International Conference on*, pages 104–109. IEEE, 2004. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1314376. → pages 5
- [29] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education*, 10(4):1–15, Nov. 2010. ISSN 19466226. doi:10.1145/1868358.1868363. URL <http://portal.acm.org/citation.cfm?doid=1868358.1868363>. → pages 2, 5
- [30] Nobuyuki Tachi, Susumu Yamazaki, and Taku Jiromaru. An Education Program on Embedded Software Development Using Open Source Hardware Aiming to Learn How to Learn New Technologies (abst.). *J.of JSEE*, 62(4):4–38, 2014. URL https://www.jstage.jst.go.jp/article/jsee/62/4/62_4_33/.pdf. → pages 11
- [31] T. Preston-Werner. How we turn \$199 Chromebooks into Ubuntu-based code learning machines for kids, Aug. 2014. URL <http://blog.codestarter.org/how-we-turn-199-chromebooks-into-ubuntu-based/>. [Last accessed 2015-05-27]. → pages 8
- [32] M. Resnick, J. Maloney, A. Monroy-Hernandez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for all. *Communications of the ACM*, 52(11), 2009. URL <http://static.elitesecurity.org/uploads/2/4/2462619/communications200911-xBSrY7lmop.pdf>. → pages 5
- [33] E. Troger, M. Brush, C. Wendling, F. Lanitz, N. Treleaven, and D. Hopf. Geany : Home Page, Oct. 2008. URL <http://www.geany.org/>. [Last accessed 2015-06-24]. → pages 11
- [34] Wilson Snyder. Intro - Verilator - Veripool, 2015. URL <http://www.veripool.org/wiki/verilator>. [Last accessed 2015-10-16]. → pages 58

Appendix A

Arduino Runtime Source

```
1 //arduinoRuntime.h
2 //Keith Lee
3 //Jan 20 ,2015
4
5 #include"arduinoRuntime.h"
6 #define AIMAGE "./aimage.bmp"
7
8
9 static int fd;
10
11
12 // Arduino Runtime main loop
13
14 int main()
15 {
16     dev_fb fb;
17     bmp bm;
18
19     setbuf(stdout,NULL);
20
21     if(fbBmp_openBmp(AIMAGE, &bm))
22     {
23         printf("\nArduinoRuntime error: Failed to initialize Arduino Image\
24             ↵ n");
25         return -1;
26     }
27     fb_init(&fb);
28
29
30     //Begin mapping to hardware
31
32     reset();
33     fb_fillScr(&fb,255,255,255);
```



```

34     fbBmp_draw(&fb, &bm, 1, OFFSET,OFFSET);
35     fd=open_devbox_io() ;
36     printf("%d\n",fd);
37     updateState(&fb, fd);
38
39     setup();
40     updateState(&fb, fd);
41     close_devbox_io(fd);
42
43     //Call user's loop function
44     while(1)
45     {
46         loop();
47         if((fd=open_devbox_io())==-1)
48         {
49             printf("ERROR: Could not open io\n");
50             return -1;
51         }
52         updateState(&fb, fd);
53         close_devbox_io(fd);
54     }
55
56     //Clean up
57
58     fb_close(&fb);
59
60
61     return 0;
62 }
63
64 void pinMode(int pin, int mode)
65 {
66
67     if(pin>13)
68         return;
69     if(mode==OUTPUT)
70     {
71         d_pin_mode [pin]=(mode<<8) |_n_output;
72         if(_n_output<10)
73             ledr[_n_output]=d_pin[pin];
74         _n_output++;
75     }
76     else
77     {
78         d_pin_mode [pin]=(mode<<8) |_n_input;
79         if(_n_input<4)
80             d_pin[pin]=keys[_n_input];
81         else
82             d_pin[pin]=switches[_n_input-4];
83         _n_input++;
84     }
85     //printf("o: %d\ni: %d\n\n",_n_output, _n_input);

```

```

86 }
87
88 void drawPinState(dev_fb* fb)
89 {
90     pixel px={PIN13_8_STARTX+OFFSET,PIN_STARTY+OFFSET};
91     unsigned char color[3];
92     int i;
93     //Set D_PINS
94     color[1]=color[2]=0;
95     for(i=13;i>=0;i--)
96     {
97         //printf("%d: %d\n", i, d_pin[i]);
98         if(d_pin[i]!=0)
99             color[0]=255;
100        else
101            color[0]=0;
102
103        fb_fillBox(fb, px, PIN_DIMX, PIN_DIMY, color[0], color[1], color
↪ [2]);
104
105        if(i == 8)
106        {
107            px.x=PIN7_0_STARTX+OFFSET;
108            px.y=PIN_STARTY+OFFSET;
109        }
110        else px.x+=PIN_GAP+PIN_DIMX;
111    }
112
113    //Set A_PINS
114    px.x=A_PINS_STARTX+OFFSET;
115    px.y=A_PINS_STARTY+OFFSET;
116    color[1]=color[2]=0;
117    for(i=5;i>=0;i--)
118    {
119        color[0]=a_pin[i];
120        fb_fillBox(fb, px, PIN_DIMX, PIN_DIMY, color[0], color[1], color
↪ [2]);
121        px.x+=PIN_GAP+PIN_DIMX;
122    }
123 }
124
125 void drawLedState(dev_fb* fb)
126 {
127     int i;
128     pixel px={LEDR_STARTX+OFFSET, LEDR_STARTY+OFFSET};
129     unsigned char color[3];
130     color[1]=color[2]=0;
131     for(i=9;i>=0;i--)
132     {
133         if(led[i])
134             color[0]=255;
135         else

```

```

136         color[0]=0;
137
138     fb_fillBox(fb, px, LEDR_DIMX, LEDR_DIMY, color[0], color[1], color
    ↪ [2]);
139     px.x+=LEDR_GAP;
140 }
141 }
142
143 void drawKeyState(dev_fb* fb)
144 {
145     int i;
146     pixel px={KEY_STARTX+OFFSET, KEY_STARTY+OFFSET};
147     unsigned char color[3];
148     color[0]=color[1]=color[2]=0;
149     for(i=3;i>=0;i--)
150     {
151         if(keys[i])
152             color[0]=255;
153         else
154             color[0]=0;
155
156         fb_fillBox(fb, px, KEY_DIMX, KEY_DIMY, color[0], color[1], color
    ↪ [2]);
157         px.x+=KEY_GAP;
158     }
159
160 }
161
162 void drawSwState(dev_fb* fb)
163 {
164     int i;
165     pixel px={SW_STARTX+OFFSET, SW_STARTY+OFFSET};
166     unsigned char top_color[3], bottom_color[3];
167     top_color[1]=top_color[2]=bottom_color[1]=bottom_color[2]=0;
168     for(i=9;i>=0;i--)
169     {
170         if(switches[i])
171         {
172             top_color[0]=255;
173             bottom_color[0]=0;
174         }
175         else
176         {
177             top_color[0]=0;
178             bottom_color[0]=255;
179         }
180         fb_fillBox(fb, px, SW_DIMX, SW_DIMY/2, top_color[0], top_color[1],
    ↪ top_color[2]);
181         px.y+=(SW_DIMY/2)+1;
182         fb_fillBox(fb, px, SW_DIMX, SW_DIMY/2, bottom_color[0],
    ↪ bottom_color[1], bottom_color[2]);
183         px.y=SW_STARTY+OFFSET;

```

```

184     px.x+=SW_GAP;
185 }
186 }
187
188 void getIo(int fd)
189 {
190     int i;
191     unsigned short input_val=devbox_get_sw(fd) & 0x3ff;
192
193     for(i=0;i<10;i++)
194     {
195         switches[i]=input_val&1;
196         input_val=input_val>>1;
197     }
198     input_val=~(devbox_get_key(fd) & 0xf);
199     for(i=0;i<4;i++)
200     {
201         keys[i]=input_val&1;
202         input_val=input_val>>1;
203     }
204
205 }
206
207 void setIo(int fd)
208 {
209     int i;
210     unsigned int output_val=0;
211     unsigned char bit_val;
212     for(i=0;i<10;i++)
213     {
214         bit_val=(ledr[i]>0)?1:0;
215         output_val=output_val|(bit_val<<i);
216     }
217     devbox_set_led(fd,output_val);
218
219 }
220
221 void setPins()
222 {
223     int i;
224     int n_input;
225     for(i=0;i<14;i++)
226     {
227         if(d_pin_mode[i]>>8==INPUT)
228         {
229             n_input=d_pin_mode[i]&0xff;
230             if(n_input<4)
231                 d_pin[i]=keys[n_input];
232             else
233                 d_pin[i]=switches[n_input-4];
234         }
235     }

```

```

236 }
237
238 void getPins ()
239 {
240     int i;
241     int n_output;
242     for (i=0;i<14;i++)
243     {
244         if (d_pin_mode[i]>>8==OUTPUT)
245         {
246             n_output=d_pin_mode[i]&0xff;
247             if (n_output<10)
248                 led[n_output]=d_pin[i];
249         }
250     }
251 }
252 }
253
254
255 void updateState (dev_fb* fb, int fd)
256 {
257     getIo (fd);
258     setPins ();
259     setIo (fd);
260     getPins ();
261     drawPinState (fb);
262     drawLedState (fb);
263     drawKeyState (fb);
264     drawSwState (fb);
265 }
266 }
267
268 void reset ()
269 {
270     int i;
271     for (i=0;i<32;i++)
272     {
273         if (i<4)
274             keys[i]=0;
275         if (i<6)
276         {
277             a_pin[i]=0;
278             seg7[i]=0;
279         }
280         if (i<10)
281         {
282             led[i]=0;
283             switches[i]=0;
284         }
285         if (i<14)
286         {
287             d_pin[i]=0;

```

```

288         d_pin_mode[i]=0;
289     }
290     gpio0[i]=0;
291 }
292 _n_input=0;
293 _n_output=0;
294 }
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312 void digitalWrite(int pin, int value)
313 {
314     if(pin<14)
315         d_pin[pin]=value;
316 }
317
318 int digitalRead(int pin)
319 {
320     return d_pin[pin];
321 }
322
323 void randomSeed(long seed)
324 {
325     srand(seed);
326 }
327
328 int a_random(int max)
329 {
330     return (rand()%max);
331 }
332
333 char lowByte(short x)
334 {
335     return x&0xff;
336 }
337
338 char highByte(short x)
339 {

```

```
340     return (x&0xff00)>>8;
341 }
342
343 char bitRead(int x, char n)
344 {
345     int bitmask=1<<n;
346     if(x&bitmask)
347         return 1;
348     else return 0;
349 }
350
351
352
353 void delay(int ms)
354 {
355     usleep(ms<<3);
356 }
357
358 void delayMicroseconds(int us)
359 {
360     usleep(us);
361 }
```

Appendix B

Node.js Server Source

```
1 var app = require('express')();
2 var http = require('http').Server(app);
3 var io = require('socket.io')(http);
4 var cp = require('child_process');
5 var fs = require('fs');
6 var cors = require('cors');
7 var run;
8 app.get('/', function(req, res){
9   res.send('<h1>DEVBOX ASYNC</h1><br> Direct access to port denied.');
```

```
10 });
11 io.on('connection', function(socket){
12   socket.on('disconnect', function(){
13     if(typeof run != "undefined") run.kill();
14     else console.log('run on connect');
```

```
15   });
16
17   socket.on('candr', function(data){
18     var _code=-1;
19     switch(data['lang'])
20     {
21     case 'c':
22     case 'arduino':
23       console.log('begin save\n');
24       fs.writeFileSync('main.c', data['code']);
25       break;
26     case 'verilog':
27       fs.writeFileSync('main.v', data['code']);
28       break;
29     default:
30       console.log('Compile: Unknown language');
```

```
31     break;
32   }
33   var mk=cp.spawn("make",[data['lang']]);
34   mk.on('error', function (err) {
```



```

35     console.log('spawn error:', err);
36     socket.emit('compile', 'There was a problem running the compiler
    ↪ . Please try again.\n');
37 });
38 mk.stdout.setEncoding('utf-8');
39 mk.stdout.on('data', function(data) {
40     socket.emit('compile', data);
41 });
42 mk.stderr.setEncoding('utf-8');
43 mk.stderr.on('data', function(data) {
44     socket.emit('compile', data);
45 });
46 mk.on('exit', function(code) {
47     console.log('MAKE exited with code '+code+'\n');
48     if(code==0) {
49         run = cp.spawn("./main");
50         run.on('error', function(err) {
51             console.log('spawn error:', err);
52             socket.emit('compile', 'There was a problem running the
    ↪ program. Please try again.\n');
53         });
54         run.stdout.setEncoding('utf-8');
55         run.stdin.setEncoding('utf-8');
56         run.stdout.on('data', function(data) {
57             socket.emit('run', data);
58         });
59         run.on('exit', function(code) {
60             socket.emit('run', '\n---program finished, returned '+code+
    ↪ '---\n');
61             run=undefined;
62         });
63     }
64     else socket.emit('compile', 'Compile failed: Cannot run');
65 });
66 });
67 socket.on('kill', function() {
68     if(typeof run != 'undefined') {
69         if(run.constructor.name == "ChildProcess") {
70             run.kill();
71         }
72     }
73 });
74
75 socket.on('error', function(error) {
76     console.log('Socket error: '+error);
77 });
78 socket.on('input', function(data) {
79     in_data=unescape(data);
80     if(typeof run != 'undefined') {
81         run.stdin.write(in_data+'\n');
82     }
83     else console.log('runtime: Not running on input');

```

```
84     });
85   });
86   var port=80;
87   if(process.argv.length>2 && !isNaN(process.argv[2]))
88     port=port+parseInt(process.argv[2]);
89   else port=port+1;
90
91
92
93 http.listen(port, function(){
94   console.log(Date.now() / 1000+' : listening on *:'+port);
95 });
```

Appendix C

Tutorials.php Source

```
1 <!DOCTYPE html>
2 <?php
3     ini_set('include_path','phplibs/');
4     include "TutoriML.php";
5     include "portnum.php";
6     if(isset($_POST['lang']))
7     {
8         $lang=$_POST['lang'];
9         $dir="tutorials/".$lang;
10        $files=array_slice(scandir($dir),2);
11    }
12    $page=$_POST['page'];
13    $tml=$_POST['tutorial'];
14    $file='main.txt';
15    $log='php_log.csv';
16    $date=new DateTime();
17    if(isset($tml))
18    {
19
20        if(isset($_POST['code']))
21        {
22            $base=$_POST['code'];
23            file_put_contents($file, $base);
24        }
25        $title=getTitle($dir, $tml);
26        $n_pages=getPages($dir, $tml);
27        if(isset($_POST['submit']))
28        {
29            $code=getCode($dir, $tml, $page);
30            $base=htmlspecialchars($base);
31            fsetCode($file, $code);
32            $base=file_get_contents($file);
33            file_put_contents($log,'Tutorial_start, '.date_timestamp_get(
                ↪ $date).';'.PHP_EOL, FILE_APPEND);
```

```

34     }
35
36     if(isset($_POST['next']) || isset($_POST['prev']))
37     {
38         $getPage="";
39
40     }
41     if(isset($_POST['next']))
42     {
43         if($page<$n_pages)
44         {
45             $page=$page+1;
46             file_put_contents($log, 'Next '.$page.', '.date_timestamp_get
                ↳ ($date).';'.PHP_EOL, FILE_APPEND);
47             $code=getCode($dir, $tml, $page);
48             fsetCode($file, $code);
49         }
50         if($page==$n_pages)
51             file_put_contents($log, 'Tutorial_end, '.date_timestamp_get(
                ↳ $date).';'.PHP_EOL, FILE_APPEND);
52         $base=file_get_contents($file);
53     }
54
55     if(isset($_POST['prev']) )
56     {
57         if($page>1){
58             $page=$page-1;
59             file_put_contents($log, 'Prev '.$page.', '.date_timestamp_get
                ↳ ($date).';'.PHP_EOL, FILE_APPEND);
60         }
61         if($page==1)
62             file_put_contents($log, 'Tutorial_first, '.date_timestamp_get
                ↳ ($date).';'.PHP_EOL, FILE_APPEND);
63         $base=file_get_contents($file);
64     }
65     $desc=getDesc($dir, $tml, $page);
66
67 }
68 else
69     file_put_contents($log, 'action, timestamp;'.PHP_EOL);
70
71
72
73
74 if(!isset($n_pages))
75     $n_pages=1;
76
77
78 function insertVars($page, $tml, $base, $lang)
79 {
80     if($base!=0)

```

```

81     echo('<input type="hidden" name="code" value="' . $base . '"></input
      ↪ >' . PHP_EOL);
82
83     echo('<input type="hidden" id="_lang" name="lang" value="' . $lang . '
      ↪ "></input>' . PHP_EOL);
84     echo('<input type="hidden" name="selected" value="true"></input>' .
      ↪ PHP_EOL);
85     echo('<input type="hidden" id="tut" name="tutorial" value="' . $tml . '
      ↪ "></input>' . PHP_EOL);
86     echo('<input type="hidden" name="page" value="' . $page . '"></input>' .
      ↪ PHP_EOL);
87 }
88 ?>
89 <html>
90   <head>
91     <script src="codemirror-4.2/lib/codemirror.js"></script>
92     <link rel="stylesheet" href="codemirror-4.2/lib/codemirror.css">
93     <link rel="stylesheet" href="codemirror-4.2/theme/eclipse.css">
94     <script src="codemirror-4.2/mode/clike/clike.js"></script>
95     <script src="codemirror-4.2/mode/verilog/verilog.js"></script>
96     <script src=socket.io.js></script>
97
98     <meta content="text/html; charset=ISO-8859-1" http-equiv="content-type"
      ↪ >
99
100    <style type="text/css">
101      .CodeMirror {border: 1px solid black}
102      div.left
103      {
104        width:49.5%;
105        height:100%;
106        float:left;
107
108      }
109      div.right
110      {
111        width:50%;
112        height:100%;
113        float:right;
114
115      }
116    </style>
117
118    <title>DEVBOX Tutorials</title>
119    <script type="text/javascript">
120
121
122    </script>
123    </head>
124    <body>
125    <font size="22"><a href="index.html"></a> TUTORIALS</font>
127 <br>
128
129 <div>
130 <form id="setlang" method="post" action="">
131     <label for="lang">Select Language: </label>
132     <select id="lang" name="lang" size=1 onchange="this.form.submit()">
133         <option disabled selected>-language-</option>
134         <option id="c" value="c">C</option>
135
136     </select>
137 </form>
138
139 <?php
140
141     if(isset($lang))
142     { ?>
143         <form id="choose" method="post" action="" name="choose">
144         <label for="tutorial">Select Tutorial: </label>
145         <?php echo('<input type="hidden" id="_lang" name="lang" value="'
146             ↪ ".$lang.'"></input>');
147         echo('<input type="hidden" id="portnum" value="' . $portnum.'"></
148             ↪ input>');
149         if(!isset($_POST['selected']))
150             echo('<select id="tutorial" name="tutorial" size="' . count(
151                 ↪ $files).">');
152         else
153             echo('<select id="tutorial" name="tutorial" size="1">');
154         foreach($files as $tmlx)
155         {
156             $file=unfile($tmlx);
157             echo('<option id="' . $tmlx.'" value="' . $tmlx.'">' . $file.' </
158                 ↪ option>');
159         }
160     ?>
161 </select>
162 <input type="hidden" name="selected" id="selected" value="true"
163     ↪ ></input>
164 <input type="hidden" name="page" id="page" value=1></input>
165
166 <button name="submit">GO</button>
167
168 <?php } ?>
169 </form>
170 </div>
171 <br>
172
173 <?php if(isset($tml))
174 {
175     ?>

```

```

173 <h1><?php echo($title);?></h1>
174 <div id="description" class="left">
175   <?php echo($desc); ?>
176   <br><br><form id="turnpage" name="turnpage" action="" method="
      ↪ post">
177     <div style="float: center; text-align: center;">
178     <?php
179       echo($page.'/'.'. $n_pages);
180     ?> </div>
181     <button name="prev" style="float: left;"> &lt;&lt; Prev</
      ↪ button>
182     <button name="next" style="float: right;"> Next &gt;&gt;</
      ↪ button>
183
184     <?php insertVars($page, $tml,$base, $lang); ?>
185   </form>
186 </div>
187 <div id="thecode" class="right">
188   <h1> Example Code </h1>
189   <form id="code_entry" method="post" action="" name="code_entry"
      ↪ accept-charset="utf-8" >
190   <textarea rows="15" id="code" name="code"><?php echo($base);
      ↪ ?></textarea>
191   <br>
192   <button name="candr" id="candr" type="button"onclick="cr(
      ↪ document.getElementById('_lang').value, document.
      ↪ getElementById('page').value);">Compile & Run</
      ↪ button>
193   <button name="kill" type="button" onclick="kp();">Stop
      ↪ Program</button>
194   <?php insertVars($page, $tml, 0, $lang); ?>
195 </form>
196
197
198 <label>Compiler Output</label><br><textarea cols="60" rows="5"
      ↪ id="comp_out" name="comp_out" readonly></textarea><br>
199 <label>Program Output</label><br><textarea cols="60" rows="5" id
      ↪ ="prog_ex" name="prog_ex" readonly></textarea><br>
200 <label>Enter your program input here</label><br><input id="input
      ↪ " type="text" cols="40"></input>
201 <button type="button" onclick="scanf(escape(document.
      ↪ getElementById('input').value));">Submit</button>
202 <br>
203
204 </div>
205 <script type="text/javascript">
206   var lang=document.getElementById("_lang");
207   var hilite="text/x-csrc"
208   switch(lang.value)
209   {
210   case "verilog":
211     hilite="text/x-verilog"

```

```

212         break;
213     case "arduino":
214     case "c":
215     default:
216         break;
217     }
218
219
220
221
222     var selected=document.getElementById("tut");
223     var optn=document.getElementById(selected.value);
224     optn.setAttribute('selected','selected');
225     selected=document.getElementById("_lang");
226     optn=document.getElementById(selected.value);
227     optn.setAttribute('selected','selected');
228
229     </script>
230 <?php } ?>
231 <script type="text/javascript">
232     var selected=document.getElementById("_lang");
233     var optn=document.getElementById(selected.value);
234     optn.setAttribute('selected','selected');
235     var text_area=document.getElementById("code");
236     var editor=CodeMirror.fromTextArea(text_area,{theme: "eclipse",
237                                     lineNumbers: true,
238                                     mode: hilite,
239                                     indentUnit: 3,
240                                     tabSize: 3,
241                                     indentWithTabs: true,
242                                     pollInterval: 0,
243                                     });
244     var portnum = document.getElementById('portnum').value;
245     var socket=io(document.domain+':'+portnum);
246     socket.connect();
247
248     socket.on('compile', function(data){
249         var c_out=document.getElementById('comp_out');
250         c_out.innerHTML+=data;
251     });
252     socket.on('run', function(data){
253         var r_out=document.getElementById('prog_ex')
254         r_out.innerHTML+=data;
255         r_out.scrollTop=r_out.scrollHeight;
256     });
257     function cr(lang, page)
258     {
259         socket.emit('candr',{lang: lang, code: editor.doc.getValue(),
260                             ↵ page: page});
261         document.getElementById('compile_out').innerHTML="";
262         document.getElementById('program_out').innerHTML="";

```



```
263
264     function kp()
265     {
266         socket.emit('kill');
267     }
268
269     function scanf(data)
270     {
271         socket.emit('input', data);
272         document.getElementById('input').value='';
273     }
274 </script>
275 </body>
276 </html>
```

Appendix D

Logic Design VHDL Assignments

D.1 2013 Cohort

**DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
EECE 353 – Digital Systems Design**

Lab 1: Combinational Logic

Lab: MCLD358

TAs: Aaron Severance, Michael Yue, Dongdong Li

In this lab, you will become familiar with the software and hardware we will be using in the labs, and use the hardware and software to implement a simple combinational logic circuit. You will also be introduced to VHDL, which is a common way of specifying digital circuits. Even though we haven't talked much about VHDL in class yet, you'll still be able to do this lab if you follow the instructions in this handout.

We will be using two pieces of software for most of this course: Quartus II, which is produced by Altera, and ModelSim, which is produced by Mentor Graphics. There are several versions of ModelSim available; we will be using one that has been modified by Altera for use with Quartus II. You can download these programs as described below (they are free), or use them on the departmental computers in the lab.

PHASE ONE: SETUP AND SIMULATION

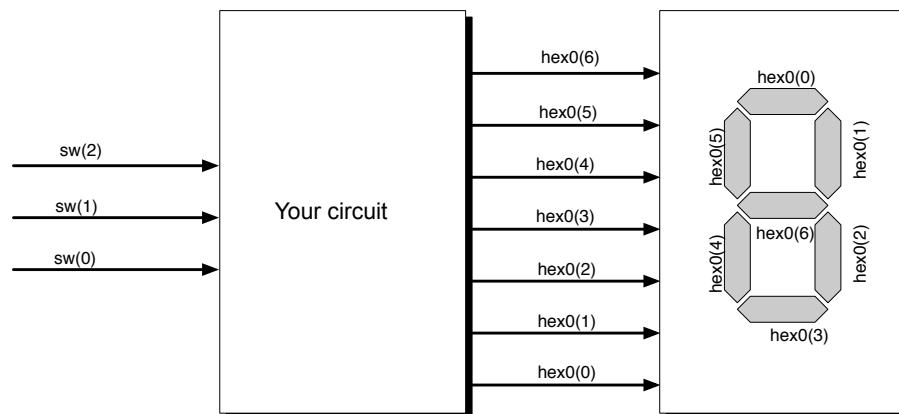
TASK 0: Install Quartus II and the Altera version of ModelSim on your home PC, laptop, or find a place you can run the software (it will be available on most of the computers available throughout the department).

- Visit <http://help.ece.ubc.ca/Altera> for download links and installation information.
- In EECE 353, we will use version 13.0 SP1 of Quartus II. This version is installed in the lab.
- If you are taking EECE 381 concurrently, you might use version 12.0 SP2 for both courses.

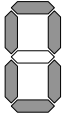

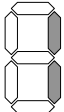

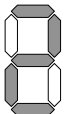

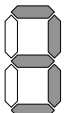

TASK 1: Download the “Digital System Design using Altera Quartus II and ModelSim” tutorial document located in the Lab 1 folder of Connect. Section 1 will help you with installing the tools. Work through Sections 2 and 3 of the tutorial document. This will introduce you to Quartus II and ModelSim, which are the tools we will use throughout the course.

TASK 2: You are going to design and simulate a combinational circuit using VHDL. Don't worry about not knowing much VHDL yet; you will be given skeleton files with most of the hard work done. You'll be guided to make some fairly minor changes to these files.

The circuit you will design is a combinational circuit with three inputs named $sw(0)$, $sw(1)$ and $sw(2)$, and seven outputs named $hex0(6)$ down to $hex0(0)$. The inputs will be connected to switches, and the output will be connected to a LED hex display as shown below.



The operation of this circuit is that it will convert a three bit binary number into seven signals that control an LED display, as shown below:

sw(2 downto 0)	Output	sw(2 downto 0)	Output
000		100	
001		101	
010		110	
011		111	

Each input to the LED (output of your circuit) controls one segment; if the input is **0**, the segment is **on** while if the input is **1**, the segment is **off** (this is called active-low signaling). The correspondence between LED input signals and the various segments in the display are shown in the diagram on the first page of this handout. So, for example, if hex(0) is a 0, then the top horizontal segment will be on.

To specify this circuit, you should download the skeleton files *converter.vhd* and *converter_tb.vhd* from the course site (in the “Labs/Lab 1” folder). The first file, *converter.vhd* is where the functionality of the block will be described. The second file, *converter_tb.vhd* is a testbench file contains the instructions the simulator will use to exercise (stimulate) your design during simulation. In this lab, the complete *converter_tb.vhd* is given to you; you won’t have to modify the testbench file. All of your changes will be in the file *converter.vhd*.

The only part of *converter.vhd* you need to modify is the part that describes the logic functionality of your block. If you look near the end of your file, you will see a block of code that looks like this:

```

case SW is
  when "000" => HEX0 <= "1000000";
  when "001" => HEX0 <= "1111001";
  when "010" => HEX0 <= "insert a string of 7 bits here";
  when "011" => HEX0 <= "insert a string of 7 bits here";
  when "100" => HEX0 <= "insert a string of 7 bits here";
  when "101" => HEX0 <= "insert a string of 7 bits here";
  when "110" => HEX0 <= "insert a string of 7 bits here";
  when others => HEX0 <= "insert a string of 7 bits here";
end case;

```

This code describes a CASE statement that is a common way of specifying combinational logic in VHDL (you might have seen CASE or SWITCH statements in software languages). This block describes how the output signal HEX0 depends SW.

The first thing to note is that SW is used as a short-form for the collection (or 3-bit *bus*) of signals SW(2), SW(1), and SW(0). When we say SW is equal to “011”, we really mean that SW(2) is equal to “0” (the first digit of “011”), SW(1) is equal to “1” (the second digit of “011”), and SW(0) is equal to “1” (the third digit of “011”). It is not a mistake that the indices count “down” through the bus (ie. the first digit is the highest numbered or *most-significant* bit); this is a common design pattern we will talk more about later. Similarly, we use HEX0 to refer to the 7-bit bus consisting of signals HEX0(6) down to HEX0(0). So, if HEX0=“1000000”, then HEX0(6) is equal to 1 (the first digit of “1000000”), HEX0(5) is equal to 0 (the second digit of “1000000” and so on).

The second thing to note is the structure of the CASE statement itself. The first line indicates the selection criteria for this CASE statement (this will become clear in the following discussion). The remaining lines in the case statement indicate an action that should occur for each value of the selection criteria. So for example, the first “when” line indicates that when the selection criteria SW is “000”, the output HEX0 should be assigned the value “1000000”. If we compare this to the second figure in this handout, we can see that when SW is 000, the output should display a pattern with the middle horizontal segment off, but all other segments on (this looks like a “0” on the display). Remembering that a segment is “off” when its control line is 1, and “on” when its control line is 0, we can see that the pattern “1000000” will turn off the middle segment (bit HEX0(6)) and turn on all the other segments (bits HEX0(5) through HEX0(0)), causing the display to show a “0”. Similarly, the second “when” line indicates that the SW is “001”, the output HEX0 should be assigned a value “1111001”. From the handout figure, this means that bits HEX0(6), HEX0(5), HEX0(4), HEX0(3), and HEX0(0) should be off, while HEX0(2) and HEX0(1) should be on. This will display a “1” on the display.

The remaining 6 lines in the case statement correspond to each of the other 6 possible values of SW. Your task is to fill in the value assigned to HEX0 for each of these 6 lines (to be more explicit, replace each string “insert a string of 7 bits here” with 7 bits corresponding to the display pattern for each line). Remember that each string of 7 bits should be surrounded by quotes, and don’t forget the semicolon at the end of each line. Note that the selection criteria for the last line is labeled “others” rather than “111”; we’ll talk more about the reasoning behind this later in the course.

Make sure you add your name and student number to the top of the *converter.vhd* file.

TASK 3: Using ModelSim, simulate your design (as you learned when you worked through Section 2 of the tutorial document). The supplied testbench *converter_tb.vhd* will apply each of the 8 possible input values to the inputs (one value every 5 ns). Using the waveform viewer, you can determine whether your design is correct. Note that there are two possible errors you might run into here. First, if you have made syntax errors, (such as forgetting the quotes or semicolons, or getting the wrong number of digits, for example) you will get error messages saying your design could not be loaded or simulated. In that case, you won’t be able to view a waveform; you have to fix the syntax error before proceeding. Once your syntax is correct, and you can simulate your design, it is possible that you see the wrong pattern of 1’s and 0’s in your output waveforms. In that case, check the assignment statements to make sure you didn’t write in the wrong values of 0’s and 1’s, or didn’t accidentally swap the order of bits within a bus. Once your simulation works, create a bit-mapped image of the simulation as described in the tutorial document.

PHASE TWO: REAL HARDWARE

In this part of the lab, you will download the circuit you designed earlier on the Altera Cyclone II. The input and output pins of this FPGA are tied to the various lights and switches and other devices.

TASK 4: Compile your design using Quartus II and be sure you don't have any synthesis errors. If you do, you will need to fix them. For this lab, it is most likely that any errors would have been uncovered during your ModelSim simulation, however, in future labs, there is a large class of errors that might be encountered here (unsynthesizable code such as "inferred latches"). If you are getting errors at this point, talk to your TA.


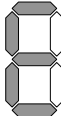






TASK 5: Read Section 4 of the tutorial document, which describes how to download your design onto the FPGA board. As described in the tutorial download your 7-segment LED Driver circuit you designed in the preparation. **BE SURE that you set the pin assignments before compiling your design (not doing so could damage the board!).** If you have any questions about how to do this, please talk to the TA.

Be prepared to demonstrate your working design for both TASK 5 and TASK 6 to the TA.

TASK 6: Extend your design so that it displays 4 different digits. To do this, you will have to add extra outputs for displays HEX3, HEX2, and HEX1 as well as extra inputs for a total of 12 switches. **DO NOT SIMPLY COPY YOUR CASE STATEMENT FOUR TIMES!** Instead, use design hierarchy to create and connect four instances of the *converter* component you developed for Task 5.

Be prepared to demonstrate your working design for both TASK 5 and TASK 6 to the TA.

TASK 7 (optional): If you finish early, play with the software and figure out what else it can do. One thing you can do is to modify the *converter.vhd* file so it displays the characters below. Next, you can extend the converter from 3 input switches to 4 input switches, allowing you to display both sets of characters, or you could display the set of 16 hexadecimal characters: 0 1 2 3 4 5 6 7 8 9 A B C D E F. The better you understand the software, the easier you will find the rest of the labs in this course.

sw(2 downto 0)	Output	sw(2 downto 0)	Output
000		100	
001		101	
010		110	
011		111	

**DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
EECE 353 – Digital Systems Design**

Lab 2: Finite State Machines

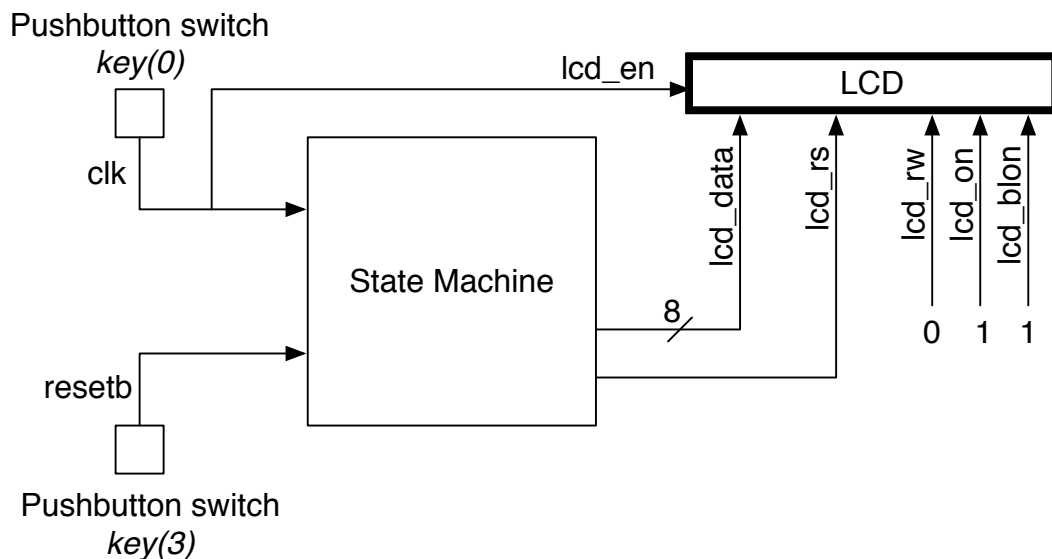
In this lab, you will create two types of finite state machines (FSMs). The first FSM will be a simple controller for the LCD display on the Altera board. The second FSM will implement the dice game of craps – it has more complex next-state logic than the LCD controller. Both FSMs will be described using VHDL.

PART 1: LCD CONTROLLER

The system you will build will cycle through and display the first five characters of your name. Each cycle, the LCD will display one character. The clock comes from pushbutton switch **key0**. Every time you depress the switch, another character appears. So if your name is “Abcde”, the LCD would display an “A” in the first cycle, add a “b” in the second cycle (so the display is "Ab"), add a “c” in the third cycle (so the display is "Abc"), etc. On the sixth cycle, it would cycle back to “A” and start again. Each character is displayed to the right of the previous characters, so after 12 cycles (for example), the LCD would display “AbcdeAbcdeAb”.

There is also a reset input; this will be controlled by the pushbutton switch **key3**. When this pushbutton switch is lowered, the system resets immediately. After a reset (and at the start), the state machine takes 6 cycles before starting with the first cycle of your name (this is due to the need to reset the LCD display; this will become clear later).

The following diagram shows the overall system you will build:



How to use the LCD:

To do this assignment, you have to know a bit about how the LCD works and how to interface with it. The LCD has five single-bit control inputs, and one 8-bit wide input bus. The five control inputs are as follows:

lcd_on :	'1' turns the LCD on. In this lab, this should always be '1'.
lcd_blon :	'1' turns the backlight on. In this lab, this should always be '1'.
lcd_rw :	whether you want to read or write to the internal registers. In this lab, we will only be writing, meaning this should always be '0'.
lcd_en :	this enable signal is used to latch in data and instructions (see below).
lcd_rs :	this tells the LCD the lcd_data bus has characters or instructions (see below).

The 8-bit bus is called **lcd_data** and is used to send instructions or characters to the LCD display.

You can communicate with the LCD using either instructions (to set the LCD in a certain mode or tell it to do something like clear the display) or using characters (in which case the character is displayed on the screen). Each cycle, you can send either one instruction or one character on the 8-bit bus. If you are sending an instruction, the **lcd_rs** signal should be set to 0, and if you are sending a character, the **lcd_rs** signal should be set to 1.

The data bus is sampled on the *falling* edge of the **lcd_en** signal. In this lab, we will drive **lcd_en** with the system clock (which comes from one of the pushbuttons). It is important to remember that the LCD instruction or character is accepted on the falling edge of this clock (this is different than the state machine, which changes states on the rising edge of the clock).

So, to be clear, to send an instruction or character, you would do the following. First, **lcd_on**, **lcd_blon**, **lcd_rw** should be fixed as described above. **lcd_en** would initially be 1. You would then drive **lcd_rs** with a 0 (if you want to send an instruction) or 1 (if you want to send a character). At the same time, you would drive either the instruction code or character code (either of which is 8 bits) on **lcd_data**. Then, **lcd_en** would drop to 0, and the LCD would either accept and execute the instruction, or accept and display the character.

There are several instructions that the LCD accepts. This handout will not describe them in detail. Instead, this handout will indicate a sequence of instructions that will set up the LCD properly. To set up the LCD, you should send the following instructions, in this order, once per cycle:

```
00111000 (hex "38")
00001100 (hex "0C")
00000001 (hex "01")
00000110 (hex "06")
10000000 (hex "80")
```

In fact, the first instruction (00111000) should be sent twice, since depending on how you implement the reset, you might miss the first one. Therefore, resetting the LCD will require 6 cycles. If you want to understand what these instructions mean, you can consult the LCD datasheet, which is on the Connect site.

Once you have set up the LCD as described above, you can send characters, one character per cycle. The following diagram shows the character encoding.

Character	Code	
	Binary	Hex
Space	00100000	20
!	00100001	21
"	00100010	22
#	00100011	23
\$	00100100	24
%	00100101	25
&	00100110	26
'	00100111	27
(00101000	28
)	00101001	29
*	00101010	2A
+	00101011	2B
,	00101100	2C
-	00101101	2D
.	00101110	2E
/	00101111	2F
0	00110000	30
1	00110001	31
2	00110010	32
3	00110011	33
4	00110100	34
5	00110101	35
6	00110110	36
7	00110111	37
8	00111000	38
9	00111001	39
:	00111010	3A
;	00111011	3B
<	00111100	3C
=	00111101	3D
>	00111110	3E
?	00111111	3F

Character	Code	
	Binary	Hex
@	01000000	40
A	01000001	41
B	01000010	42
C	01000011	43
D	01000100	44
E	01000101	45
F	01000110	46
G	01000111	47
H	01001000	48
I	01001001	49
J	01001010	4A
K	01001011	4B
L	01001100	4C
M	01001101	4D
N	01001110	4E
O	01001111	4F
P	01010000	50
Q	01010001	51
R	01010010	52
S	01010011	53
T	01010100	54
U	01010101	55
V	01010110	56
W	01010111	57
X	01011000	58
Y	01011001	59
Z	01011010	5A
[01011011	5B
¥	01011100	5C
]	01011101	5D
^	01011110	5E
_	01011111	5F

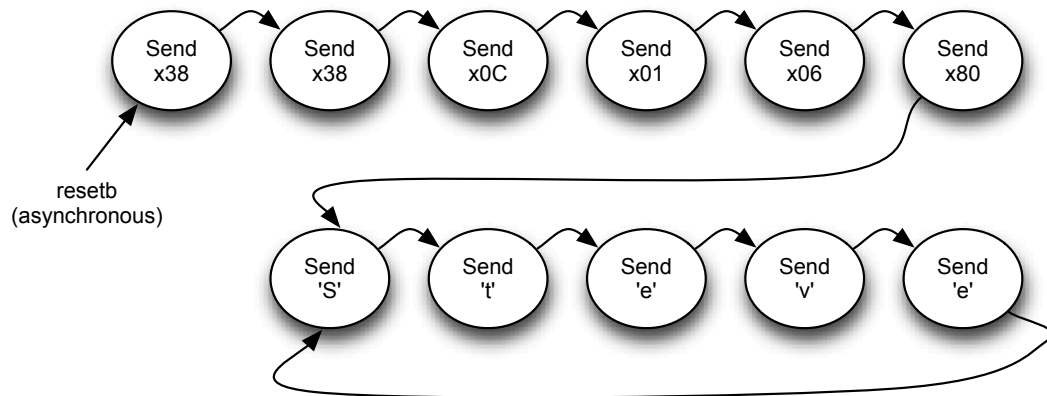
Character	Code	
	Binary	Hex
`	01100000	60
a	01100001	61
b	01100010	62
c	01100011	63
d	01100100	64
e	01100101	65
f	01100110	66
g	01100111	67
h	01101000	68
i	01101001	69
j	01101010	6A
k	01101011	6B
l	01101100	6C
m	01101101	6D
n	01101110	6E
o	01101111	6F
p	01110000	70
q	01110001	71
r	01110010	72
s	01110011	73
t	01110100	74
u	01110101	75
v	01110110	76
w	01110111	77
x	01111000	78
y	01111001	79
z	01111010	7A
(01111011	7B
	01111100	7C
)	01111101	7D
→	01111110	7E
←	01111111	7F

So, for example, if you wanted to display an "a", you would send 01100001 on the **lcd_data** bus. Note that the table above includes both binary and hexadecimal (base-16) for each code; computer engineers like to talk in hexadecimal, since it is more convenient than binary. Other characters are available, and you can even design your own characters. See the datasheet on the web site if you want more information.

There are stringent timing requirements that must be met using the LCD. However, in this lab, we are using the pushbutton switch as a clock, and it is not possible for you to push the button so fast that you are in danger of violating any of these minimum times. All that matters for this lab is that you need to make sure that the control lines are steady when the clock (**led_en**) switches from high to low.

If you want more information on the LCD, see the datasheet on Connect.

Design a state machine to implement the circuit as described on the first page of this handout. The state diagram might be something like this (if your name is "Steve"):



Upon reset, the state machine cycles through the first six states regardless of the input. The reset is asynchronous; review the course notes to make sure you remember what this means. The reset is also active low, meaning that a "0" means reset, and a "1" means normal operation (this makes it easier to use the pushbutton switch). The state machine is positive-edge triggered; this means that the transition from one state to the next occurs on the rising edge of the clock. The outputs of the state machine are the signals `lcd_rs` and `lcd_data`; given the discussion on the previous pages, you should be able to figure out what should be driven on these signals each cycle. Note that this is a Moore state machine, meaning the output depends only on the current state.

Simulate your design using ModelSim, and make sure that it works as expected, before coming into the lab. A very simple testbench is provided on Connect. This testbench resets the system, and toggles the clock. The clock cycle in this testbench is set to 6 ns, so I would suggest selecting a run-time length of 80 ns to ensure you see all the state transitions. Don't forget to Zoom Full to see the whole thing, however, you'll probably have to Zoom in to see enough detail to convince you it is working. Manually observe the waveform and make sure it matches what you expect.

Hint: Earlier I mentioned that the LCD accepts data on the falling edge of the clock. Don't be confused. In the state machine you design here, the state changes (and hence output changes) all happen on the rising clock edge. This is a normal state machine, just like we discuss in class.

Testing on the DE2

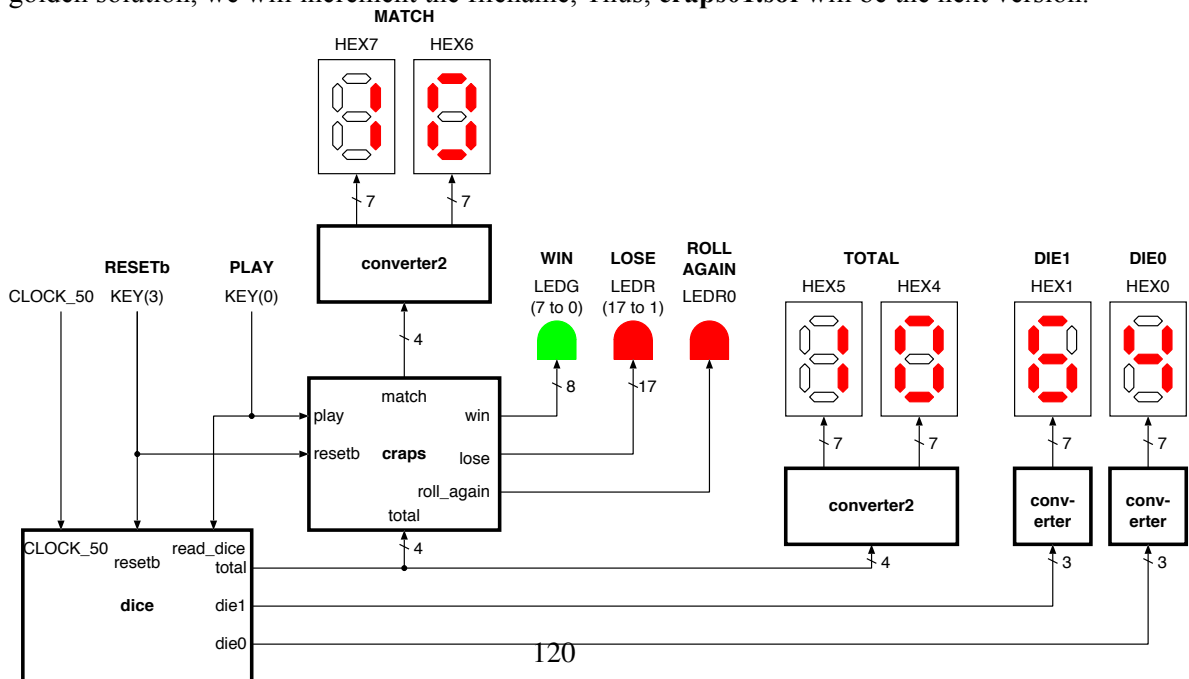
1. Download the circuit you designed onto your DE2 board. **Again, remember to use the pin assignments file from Lab 1.** Cycle through the states and show that it operates as expected. Test the reset button to make sure that works too. You will probably find it easier to see what is going on by wiring the state bits (probably called something like "present_state" or "current_state" in your VHDL code) to the green LEDs so you can easily see what state you are in. Interestingly, this highlights a fundamental limitation of debugging directly on hardware -- you can not automatically see any of the internal signals unless you have manually connected them to an output before compilation.
2. Be prepared to demonstrate a working LCD design to the TA in the lab during marking. The TA may ask you to make small changes to your VHDL code. You may wish to make the modification in either ModelSim or Quartus II, but you will need to demonstrate the working design on the DE2 board to earn full performance marks. Note that the TA will ask to see your code, and will ask questions about it. You should be able to answer these questions to get full marks.

PART 2: CRAPS DICE GAME

You will build a game controller **craps.vhd** and two types of 7-segment display converters to play the dice game of craps. The **dice.vhd** module is provided to you. Game play is described below.

1. Initially, display the initial dice values as '--' on the 7-segment displays. Light up LEDR0 to indicate the user can press the play button to roll the dice.
2. The player is making his first roll. He presses **key0** to roll both dice, with values 1 to 6 of each die appearing on HEX1 and HEX0 and the total appearing on HEX5 and HEX4. When **key0** is released, evaluate the total of the two dice as follows:
 - a. If the player rolls a 7 or 11, the player wins. Light up the green LEDs and go to step 4.
 - b. If the player rolls a 2, 3 or 12, the player loses. Light up red LEDs(17..1) and go to step 4.
 - c. Since this is the first roll for the player, the total value of the dice is the new target to match. Display this matching value on HEX7 and HEX6. Light up LEDR0 to indicate roll again, and go to step 3.
3. The player is now on his second or further rolls. Like step 2, the player presses **key0** to roll both dice, display the dice values and their total on HEX3 to HEX0. When **key0** is released, evaluate the total of the two dice as follows:
 - a. If the player rolls a 7, the player wins. Light up the green LEDs and go to step 4.
 - b. If the player total matches the target value memorized from step 2, the player wins. Light up the green LEDs and go to step 4.
 - c. Otherwise, the player has neither won nor lost but must roll again. Light up LEDR0 to indicate roll again and restart this step.
4. The game is over. Be sure to keep indicating win or lose using the red or green LEDs, and leave the last dice, total, and match values so the user can confirm the win or loss. Further presses of **key0** should not change the LEDs or 7-segment displays. LEDR0 should be off. The player must press **key3** to reset the game and return to step 1.

This part is much harder than it appears. Start early. You will have to practice getting everything right with clocks, sensitivity lists, reset, and gameplay. On Connect, we are providing a golden solution in **craps00.sof** which you can download to your board. Your solution must match the gameplay of this solution exactly; if you are not sure how something should work, try it out. Note: if we have to correct the golden solution, we will increment the filename, Thus, **craps01.sof** will be the next version.



D.2 2014 Cohort

**DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
EECE 353 – Digital Systems Design**

Lab 2: Finite State Machines

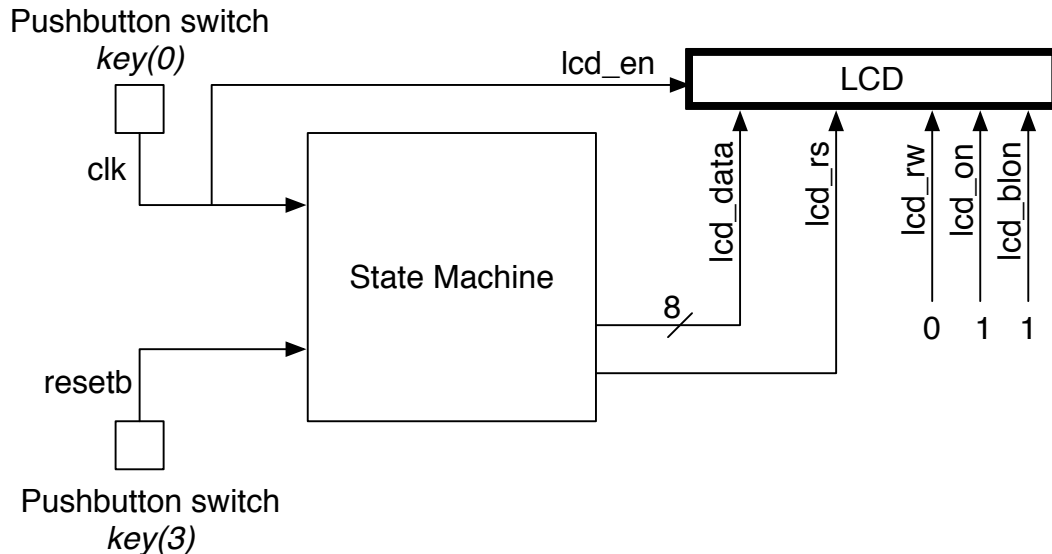
In this lab, you will create two types of finite state machines (FSMs). The first FSM will be a simple controller for the LCD display on the Altera board. The second FSM will fill the VGA screen with colours using a VGA controller. Both FSMs will be described using VHDL.

WEEK 1: LCD CONTROLLER

The first system you will build will cycle through and display the first five characters of your name. Each cycle, the LCD will display one character. The clock comes from pushbutton switch **key0**. Every time you depress the switch, another character appears. So if your name is "Abcde", the LCD would display an "A" in the first cycle, add a "b" in the second cycle (so the display is "Ab"), add a "c" in the third cycle (so the display is "Abc"), etc. On the sixth cycle, it would cycle back to "A" and start again. Each character is displayed to the right of the previous characters, so after 12 cycles (for example), the LCD would display "AbcdeAbcdeAb".

There is also a reset input; this will be controlled by the pushbutton switch **key3**. When this pushbutton switch is lowered, the system resets immediately. After a reset (and at the start), the state machine takes 6 cycles before starting with the first cycle of your name (this is due to the need to reset the LCD display; this will become clear later).

The following diagram shows the overall system you will build:



How to use the LCD:

To do this assignment, you have to know a bit about how the LCD works and how to interface with it. The LCD has five single-bit control inputs, and one 8-bit wide input bus. The five control inputs are as follows:

lcd_on :	'1' turns the LCD on. In this lab, this should always be '1'.
lcd_blon :	'1' turns the backlight on. In this lab, this should always be '1'.
lcd_rw :	whether you want to read or write to the internal registers. In this lab, we will only be writing, meaning this should always be '0'.
lcd_en :	this enable signal is used to latch in data and instructions (see below).
lcd_rs :	this tells the LCD the lcd_data bus has characters or instructions (see below).

The 8-bit bus is called **lcd_data** and is used to send instructions or characters to the LCD display.

You can communicate with the LCD using either instructions (to set the LCD in a certain mode or tell it to do something like clear the display) or using characters (in which case the character is displayed on the screen). Each cycle, you can send either one instruction or one character on the 8-bit bus. If you are sending an instruction, the **lcd_rs** signal should be set to 0, and if you are sending a character, the **lcd_rs** signal should be set to 1.

The data bus is sampled on the *falling* edge of the **lcd_en** signal. In this lab, we will drive **lcd_en** with the system clock (which comes from one of the pushbuttons). It is important to remember that the LCD instruction or character is accepted on the falling edge of this clock (this is different than the state machine, which changes states on the rising edge of the clock).

So, to be clear, to send an instruction or character, you would do the following. First, **lcd_on**, **lcd_blon**, **lcd_rw** should be fixed as described above. **lcd_en** would initially be 1. You would then drive **lcd_rs** with a 0 (if you want to send an instruction) or 1 (if you want to send a character). At the same time, you would drive either the instruction code or character code (either of which is 8 bits) on **lcd_data**. Then, **lcd_en** would drop to 0, and the LCD would either accept and execute the instruction, or accept and display the character.

There are several instructions that the LCD accepts. This handout will not describe them in detail. Instead, this handout will indicate a sequence of instructions that will set up the LCD properly. To set up the LCD, you should send the following instructions, in this order, once per cycle:

```
00111000 (hex "38")
00001100 (hex "0C")
00000001 (hex "01")
00000110 (hex "06")
10000000 (hex "80")
```

In fact, the first instruction (00111000) should be sent twice, since depending on how you implement the reset, you might miss the first one. Therefore, resetting the LCD will require 6 cycles. If you want to understand what these instructions mean, you can consult the LCD datasheet, which is on the Connect site.

Once you have set up the LCD as described above, you can send characters, one character per cycle. The following diagram shows the character encoding.

Character	Code	
	Binary	Hex
Space	00100000	20
!	00100001	21
"	00100010	22
#	00100011	23
\$	00100100	24
%	00100101	25
&	00100110	26
'	00100111	27
(00101000	28
)	00101001	29
*	00101010	2A
+	00101011	2B
,	00101100	2C
-	00101101	2D
.	00101110	2E
/	00101111	2F
0	00110000	30
1	00110001	31
2	00110010	32
3	00110011	33
4	00110100	34
5	00110101	35
6	00110110	36
7	00110111	37
8	00111000	38
9	00111001	39
:	00111010	3A
;	00111011	3B
<	00111100	3C
=	00111101	3D
>	00111110	3E
?	00111111	3F

Character	Code	
	Binary	Hex
@	01000000	40
A	01000001	41
B	01000010	42
C	01000011	43
D	01000100	44
E	01000101	45
F	01000110	46
G	01000111	47
H	01001000	48
I	01001001	49
J	01001010	4A
K	01001011	4B
L	01001100	4C
M	01001101	4D
N	01001110	4E
O	01001111	4F
P	01010000	50
Q	01010001	51
R	01010010	52
S	01010011	53
T	01010100	54
U	01010101	55
V	01010110	56
W	01010111	57
X	01011000	58
Y	01011001	59
Z	01011010	5A
[01011011	5B
¥	01011100	5C
]	01011101	5D
^	01011110	5E
_	01011111	5F

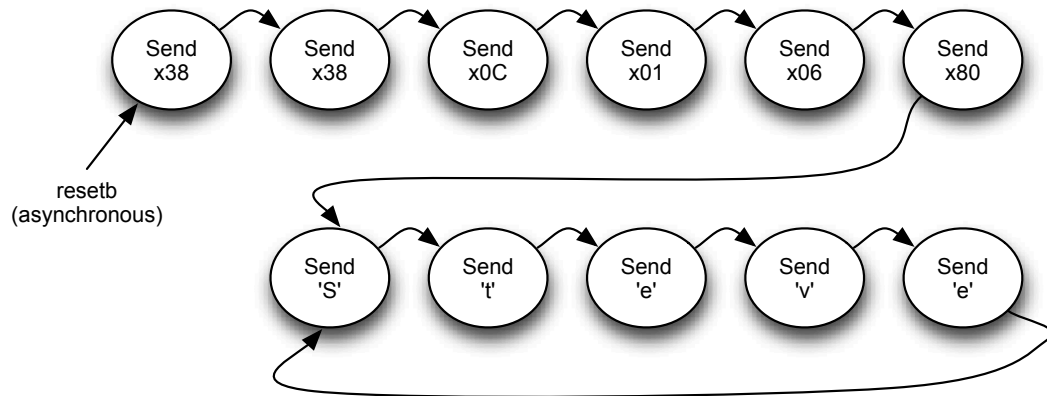
Character	Code	
	Binary	Hex
`	01100000	60
a	01100001	61
b	01100010	62
c	01100011	63
d	01100100	64
e	01100101	65
f	01100110	66
g	01100111	67
h	01101000	68
i	01101001	69
j	01101010	6A
k	01101011	6B
l	01101100	6C
m	01101101	6D
n	01101110	6E
o	01101111	6F
p	01110000	70
q	01110001	71
r	01110010	72
s	01110011	73
t	01110100	74
u	01110101	75
v	01110110	76
w	01110111	77
x	01111000	78
y	01111001	79
z	01111010	7A
(01111011	7B
	01111100	7C
)	01111101	7D
→	01111110	7E
←	01111111	7F

So, for example, if you wanted to display an "a", you would send 01100001 on the **lcd_data** bus. Note that the table above includes both binary and hexadecimal (base-16) for each code; computer engineers like to talk in hexadecimal, since it is more convenient than binary. Other characters are available, and you can even design your own characters. See the datasheet on the web site if you want more information.

There are stringent timing requirements that must be met using the LCD. However, in this lab, we are using the pushbutton switch as a clock, and it is not possible for you to push the button so fast that you are in danger of violating any of these minimum times. All that matters for this lab is that you need to make sure that the control lines are steady when the clock (**led_en**) switches from high to low.

If you want more information on the LCD, see the datasheet on Connect.

Design a state machine to implement the circuit as described on the first page of this handout. The state diagram might be something like this (if your name is "Steve"):



Upon reset, the state machine cycles through the first six states regardless of the input. The reset is asynchronous; review the course notes to make sure you remember what this means. The reset is also active low, meaning that a "0" means reset, and a "1" means normal operation (this makes it easier to use the pushbutton switch). The state machine is positive-edge triggered; this means that the transition from one state to the next occurs on the rising edge of the clock. The outputs of the state machine are the signals `lcd_rs` and `lcd_data`; given the discussion on the previous pages, you should be able to figure out what should be driven on these signals each cycle. Note that this is a Moore state machine, meaning the output depends only on the current state.

Simulate your design using ModelSim, and make sure that it works as expected, before coming into the lab. A very simple testbench is provided on Connect. This testbench resets the system, and toggles the clock. The clock cycle in this testbench is set to 6 ns, so I would suggest selecting a run-time length of 80 ns to ensure you see all the state transitions. Don't forget to Zoom Full to see the whole thing, however, you'll probably have to Zoom in to see enough detail to convince you it is working. Manually observe the waveform and make sure it matches what you expect.

Hint: Earlier I mentioned that the LCD accepts data on the falling edge of the clock. Don't be confused. In the state machine you design here, the state changes (and hence output changes) all happen on the rising clock edge. This is a normal state machine, just like we discuss in class.

Testing on the DE2

1. Download the circuit you designed onto your DE2 board. **Again, remember to use the pin assignments file from Lab 1.** Cycle through the states and show that it operates as expected. Test the reset button to make sure that works too. You will probably find it easier to see what is going on by wiring the state bits (probably called something like "present_state" or "current_state" in your VHDL code) to the green LEDs so you can easily see what state you are in. Interestingly, this highlights a fundamental limitation of debugging directly on hardware -- you can not automatically see any of the internal signals unless you have manually connected them to an output before compilation.
2. In the first week, you must get the LCD controller fully working. The TA will give you up to 5 marks for the correct operation of your LCD controller.

WEEK 2: VGA CONTROLLER

In this part of the lab, you will get more experience creating state machines. Compared to Week 1, the FSM you will implement will be slightly more complex. You will also learn how to use an embedded components that we will give you; this is common practice in industrial design – taking predesigned components that are either purchased or written by another group and incorporating them into your own design. The embedded component we will give you is a VGA Adapter core, which continuously draws to a VGA screen. You will design a VGA PixelWriter. It tells the VGA Adapter core to change the colour of specific pixels. Together, the PixelWriter and Adapter Core form the VGA Controller.

Note: those of you taking EECE 381 – in that course, you are using C software to write to a VGA display. This is fundamentally different, since here you will use *hardware* to interface to a very different sort of VGA core. If you are *not* taking EECE 381 don't worry, this is different enough that you are not at a disadvantage.

The top level diagram of your lab, `lab2vga.vhd`, is as follows. The VGA Adapter core is the part given to you, so all the excitement will be in the block labeled “your circuit”. This handout first explains how to use the VGA Adapter core, and then specifies what your circuit should do.

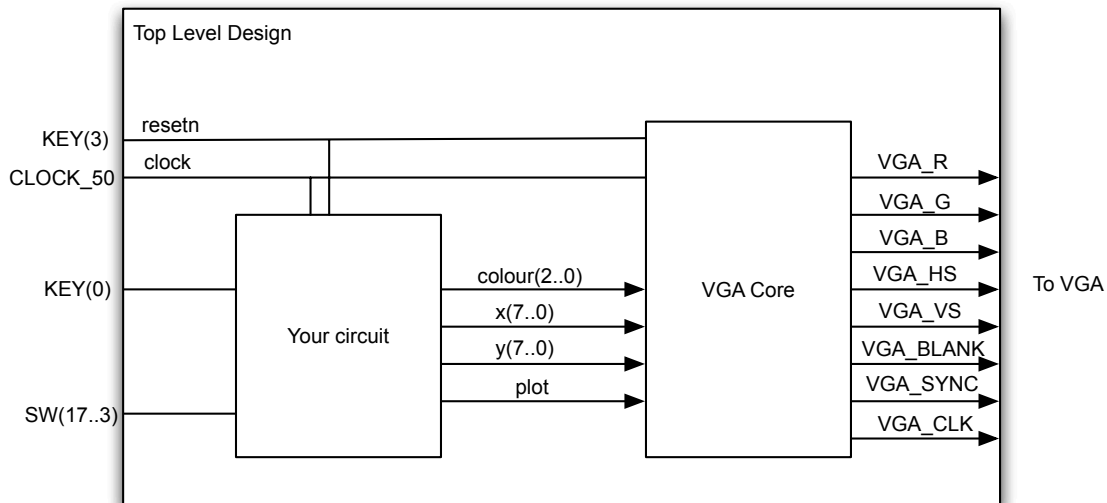


Figure 1: Overall block diagram. The top-level file is `lab2vga.vhd`.

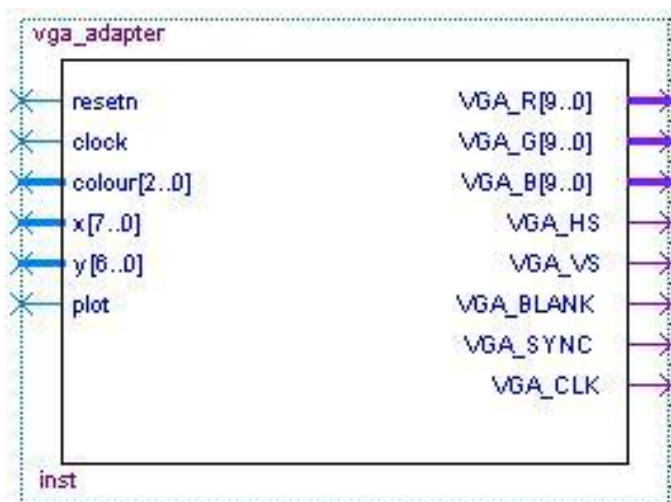
Task 1: Understand the VGA Adapter Core

The VGA Adapter core was created at the University of Toronto for a course similar to EECE 353. The following describes enough for you to use the core; more details can be found on University of Toronto's web page: http://www.eecg.utoronto.ca/~jayar/ece241_07F/vga

Some of the following figures have been taken from that website (with permission!).

In order to save on the limited memory on DE2 board, the VGA Adapter core has been setup to display a grid of 160x120 pixels, with the interface shown in Figure 2:

Inputs
From your circuit
(You will be working
with these)



Outputs
To DAC & Monitor
(You do not need to
control these)

Figure 2: VGA Adapter core as a black box

Inputs:

Resetn	Active low reset signal. Digital circuits with state elements should always contain a reset.
Clock	Clock signal. The VGA Adapter core must be fed with a 50MHz clock to function correctly.
colour(2 downto 0)	Pixel colour (3 bits). Sets the colour of the pixel to be drawn. The three bits indicate the presence of Red, Green and Blue components for a total of 8 colour combinations.
x(7 downto 0)	X coordinate of pixel to be drawn (8 bits) – supported values $0 \leq x < 160$.
y(6 downto 0)	Y coordinate of pixel to be drawn (7 bits) – supported values $0 \leq y < 120$.
Plot	Active high plot signal. Raise this signal to cause the pixel at (x,y) to be set to the specified colour on the next rising clock edge.

Outputs:

VGA_CLK	VGA clock signal.
VGA_R(9 downto 0) VGA_G(9 downto 0) VGA_B(9 downto 0)	Red, Green, Blue components of display (10 bits). These signals are connected to the Digital-to-Analog Converter (DAC) on the DE2 board before transmitting to the monitor.
VGA_HS VGA_VS VGA_SYNC VGA_BLANK	VGA control signals.

Note that you will connect the outputs of the VGA Adapter core directly to appropriate output pins of the FPGA.

You can picture the VGA screen as a grid of pixels shown in Figure 3. The X/Y position (0,0) is located on the top-left corner and (159,119) pixel located at the bottom-right corner. The role of the VGA Adapter core is to continuously draw **the same thing on the screen** at a regular rate, eg 60 Hz. To do this, it has an internal memory that stores the colour of each pixel. Your VGA PixelWriter will write new pixel colours to the VGA Adapter core, such that on the next screen refresh it will display a slightly different screen.

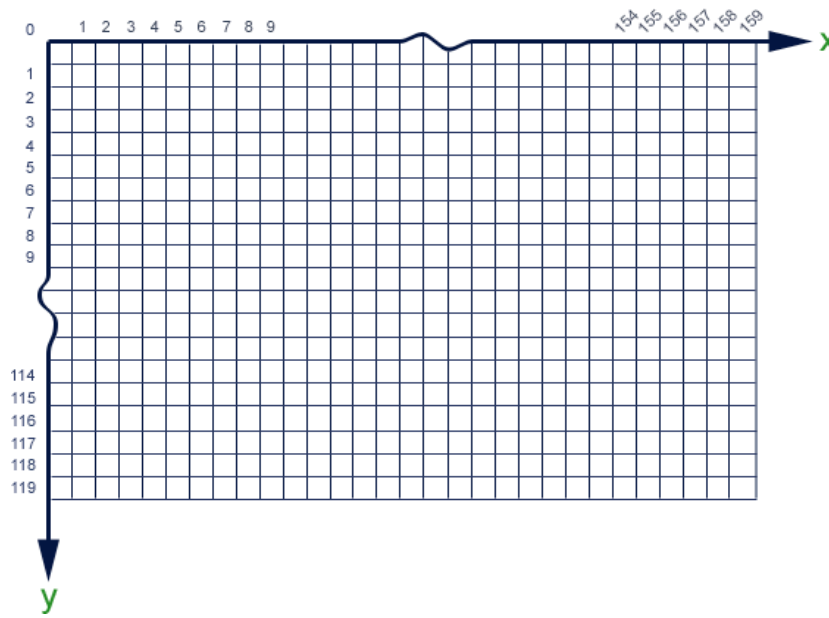


Figure 3: VGA Adapter core's display grid

To set the colour of a pixel, you drive the x input with the x position of the pixel, drive the y input with the y position of the pixel, and colour with the colour you wish to use. You then raise input plot. At the next rising clock edge, the pixel colour is accepted by the VGA Adapter core's memory. Starting on the next screen redraw, the pixel will take on the new colour.

In the following timing diagram (from the UofT Website), two pixels are changed: one at (15, 62) and the other at (109,12). As you can see, the first pixel drawn is red and is placed at (15, 62). The second is a yellow pixel at (109, 12). It is important to note that, at most, *one pixel can be changed on each cycle*. Thus, if you want to change the colour of m pixels, you need m cycles.

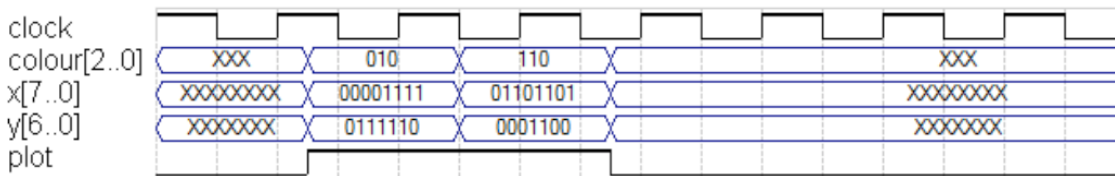


Figure 4: Timing Diagram

The source code for the VGA Adapter core consists of multiple files and is provided on the EECE 353 Connect site. This core is written in Verilog, not VHDL. That is ok: when making a structural description, you can include Verilog modules just as you can VHDL modules (this “mixed language” design approach is common in industry).

The Verilog files describing the VGA Adapter core can be included into Altera Quartus II project just like the VHDL files. If you read them, you would notice that the “module” definition in Verilog is similar to “entity” in VHDL. This means you can instantiate modules in the Verilog files as components in your VHDL files.

Task 1: Get VGA Adapter Core to Work

To help you understand the VGA Adapter core, we have created a `vga_demo.vhd` file. This file does nothing but connect the VGA Adapter core I/O to switches so you can experiment. We suggest you download this file, understand it, and try it out. It will help you understand how the inputs of the core work.

Note that this file is *only* used for you to understand the core; you will *not* use it when constructing your circuit for this lab. This task is not worth any marks, but you should do it to ensure that everything else is working before starting the main task below.

Task 2: Fill the Screen

You will create a new component that acts as a VGA PixelWriter. It will contain a simple FSM to fill the screen with colours. This is done by writing to one pixel at a time in the VGA Adapter core. Each row will be set to a different colour (repeating every 8 rows).

Since you can only set one pixel at a time, you will need a FSM that does something like this:

```
        for y = 0 to 119 {
            for x = 0 to 159 {
                set pixel (x, y) to colour ( y mod 8)
            }
        }
```

Create an FSM that implements the above algorithm. The top-level file `lab2vga.vhd` is available on the Connect site. Your design should have an asynchronous reset which will be driven by `KEY(3)`. You don't need to use `KEY(0)` or any of the switches in this task. Note that your circuit will be clocked by `CLOCK_50` (this is different than week 1, where you were clocking using a key).

Test your design on the DE2. You need your DE2 board with a USB cable, a VGA cable, and a VGA-capable display. In the MCLD358 lab, a VGA cable and LCD display are provided. Download the lab 4 files containing: `vga_adapter.v`, `vga_controller.v`, `vga_address_translator.v` and `vga_demo.vhd` from the Connect site. Add these files to a new Quartus II project, import your pin assignments, then compile and program the design onto your DE2 board.

Use the VGA cable to connect your DE2 board to the VGA display. Most new LCD displays have multiple inputs, including DVI (digital) and VGA (analog). The LCD displays in MCLD358 use the DVI input for the PC, while the VGA input is connected to a VGA cable for you to use. You can switch between the inputs using the display's input select button. Note: the VGA connection on your laptop is an **OUTPUT**, so **do not connect your laptop's VGA port to your DE2 board**.

Hint: Modelsim will be very useful for debugging. Start by looking at your x and y counters.

Task 3: Challenge Circuit

Modify your circuit to draw a 4x4 square at an `<x,y>` location specified by the switches. When the switches change, erase the whole screen (draw it black), then draw the square.

Technical grading: 2 marks (setting colour of 1 or more lines, single colour)
2 marks (setting entire screen colour, varying colours)
1 mark (challenge circuit)



Laboratory 3 - Digital Systems Design

Learning Objectives

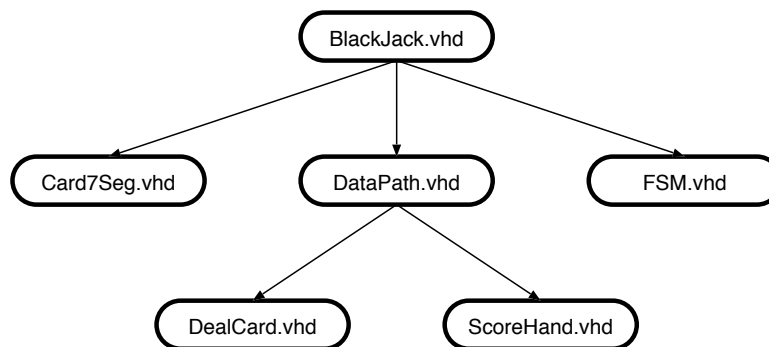
1. Design a non-trivial finite state machine and express it in VHDL.
2. Build complex combinational functions in VHDL.
3. Generate random numbers in hardware.
4. Interact with a clock.

Abstract

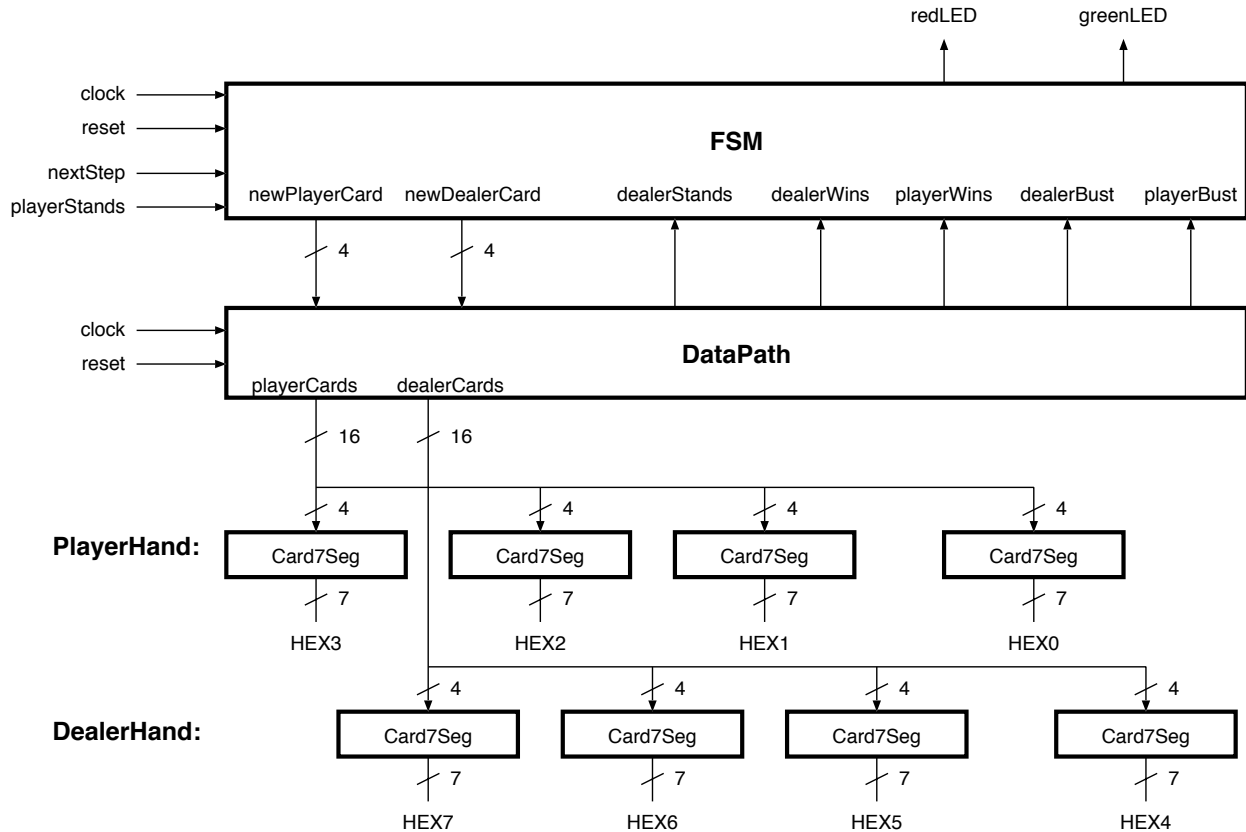
In this lab, you will design a one-player Blackjack card game using your DE2 board. The main part will be a complex FSM used to control the sequence of dealing cards to one player and the dealer. You will also develop a combinational function to drive the 7-segment display to indicate Jack, Queen, King and Ace cards, and a combinational function to count the value of a hand, where Aces can be either 1 or 11. As well, you will have a sequential part that determines the next card to be drawn. To solve this problem, you will use a variety of techniques in programming with VHDL: FSMs, combinational logic, and variables.

Organization

You will create a single design, consisting of multiple VHDL files. The top-level filename you must use is **blackjack.vhd**, with lower-level entities organized as described below.



The top-level file will be provided to you. You must create the other files. A block diagram of the top-level design, which will be created inside **blackjack.vhd**, is shown below.



Requirements

The functional specification of the lab requirements is as follows:

- Each card is represented by a 4-bit value. Ignore suites (hearts, diamonds, clubs, spades). Each value indicates:
 - 0 is “no card”
 - 2 through 10 are themselves
 - 1 is Ace, 11 is Jack, 12 is Queen, and 13 is King
 - 14, 15 are not used
- The dealer and player can each hold a hand of up to 4 cards. No more than 4 cards are allowed per player. To remember an entire hand, you will need 16 bits. You will need separate register memories for both player and dealer hands.
- Display the cards in a hand on the 7-segment display. At all times:
 - Hex3-Hex0: display the player hand
 - Hex7-Hex4: display the dealer hand
- Each card is a 4-digit number that is displayed as a single digit on the 7-segment display. Write **card7seg.vhd** to convert a 4-bit card value into an appropriate LED display pattern as follows:
 - The value 0 is “no card” and should be displayed as a blank (all LEDs off)
 - Ace as “A”, 10 as “0”, Jack as “J”, Queen as “q”, and King as “H”
 - 2 through 9 as themselves, making sure the numeral 9 appears differently than “q”
- As a player, during your turn you can request either “hit” for another card, or “stand” for no more cards.

- SW0 = '0' indicates "hit". SW0 = '1' indicates "stand".
6. The dealer must always follow the house rules (regardless of the player's hand):
 - If the current dealer hand totals 16 or less, the dealer takes another card
 - If the current dealer hand totals 17 or more, the dealer stands
 7. Counting a hand is done in **scorehand.vhd**. You will need two instances of this component, one to count the dealer hand and one to count the player hand. This component takes a complete hand (16-bit value) as inputs, counts the cards, and produces a 5-bit total for the output. There are two additional 1-bit outputs: "stand" indicates the hand ≥ 17 , and "bust" indicates the hand > 21 . This routine must be written entirely in combinational logic (with no clock). To simplify things, the use of VARIABLE types is suggested. When counting, use a larger number of bits internally (6 bits is enough to represent the maximum possible sum of 44), but the final result should be reduced to 5 bits. Remember that each Ace takes on the value of '1' or '11', as needed, to give the maximum possible hand value that is still less than or equal to 21. **The logic for counting is complex, so make sure you give yourself enough time to think about it and do plenty of testing!** **The use of a testbench and ModelSim is strongly encouraged.**
 - The Jack, Queen, and King each represent a value of 10.
 - Each Ace represents the value of 11 or 1, individually as needed, to give a maximum possible hand total value that is less than or equal to 21.
 - If the final total hand value is ≥ 17 , the **stand** output should be '1'.
 - If the final total hand value > 21 , the hand is bust. In this case, force the hand value to 31 (all outputs will be '1'; this helps simplify testing if you are displaying on LEDs).
 8. All of your sequential logic (FSMs, registers) **must use CLOCK_50 as the only clock**. **Use KEY0 as a combinational input** where pressing the key tells the game to take the "next step". As KEY0 is pressed, the game makes forward progress. Your FSM may sequence through several states (clock cycles) to accomplish the current step, and then pause until the user presses KEY0 again. Usually, KEY0 means a new card will be given out (either to the dealer or the player). *Hint:* You may find it easier to design a small FSM that waits for KEY0 to go from 1 to 0, and in response emits a pulse that is only a single clock cycle wide.
 9. The required sequence of steps are:
 - a. **Start.** Give player a card.
 - b. Give dealer a card.
 - c. Give player a second card. If player stands, go to **DealersTurn**.
 - d. Give player a third card. If player stands, or goes bust, go to **DealersTurn**.
 - e. Give player a fourth card.
 - f. **DealersTurn.** Give dealer a second card. If dealer stands, go to **Winner**.
 - g. Give dealer a third card. If dealer stands, or goes bust, go to **Winner**.
 - h. Give dealer a fourth card.
 - i. **Winner.** Decide winner:
 - If both go bust, there is no winner, go to **EndGame**.
 - If dealer \geq player, or player is bust, go to **DealerWins**.
 - If dealer $<$ player, or dealer is bust, go to **PlayerWins**.
 - j. **DealerWins.** Turn on LEDR[17:0]. Go to **EndGame**.
 - k. **PlayerWins.** Turn on LEDG[7:0]. Go to **EndGame**.
 - l. **EndGame.** Wait forever.

10. To give the dealer or player a new random card, we will use a few simple tricks. First, assume we are dealing from an infinite deck, so it is equally likely to generate any card, no matter which cards have already been given out. Second, assume that when the player presses the “next step” key, an unpredictable amount of time has passed, representing a random delay. During this random delay interval, your **dealcard.vhd** should be continuously counting from the first card (Ace=1) to the last card (King=13), and then wrapping around to Ace=1 at a very high rate (eg, 50MHz). To obtain a random card, we simply sample the current value of this counter during one clock cycle after a random delay, ie when the user presses “next step”.
11. Use KEY3 to reset the game and go to **Start** at any point.

DO NOT USE KEY0 AS A CLOCK SIGNAL! Your design must use only CLOCK_50 as the clock that controls all memory elements and advances you from state to state. **You will lose marks if you do not follow this instruction.**

Week 1 Tasks

During your first week, you will get marks as follows:

- Datapath Part (3 marks)
 - 1 mark for creation and testing of **card7seg.vhd**
 - 1 mark for creation and testing of **dealcard.vhd**
 - 1 mark for creation and testing of a simple version of **scorehand.vhd**, where you assume an Ace=10/Jack=11/Queen=12/King=13.
- FSM Part (2 marks)
 - 2 marks for creation and testing of a **simple** version of **fsm.vhd** that deals only 2 random cards per player. The simple FSM should not worry about a winner, or a hand going bust. It must wait for KEY0 to be pressed once to deal each card, otherwise you will not get a random card. It should also respond to KEY3, which will reset the game. Note: to simplify TA marking, please do exactly as requested.

Week 2 Tasks

The changes from Week 1 should be obvious, but to highlight the requirements once more:

- (2 marks) Complete **scorehand.vhd** to treat Jack/Queen/King all as 10 and its treatment of each Ace to be either 1 or 11, as required, to get a maximum score of 21 without going bust. Note that a hand containing two Aces could treat one Ace=1 and another Ace=11 to get a perfect score of 21, whereas a hand with two Aces and a 10 would treat both Ace=1 to get a score of 12.
- (1 mark) Create **datapath.vhd** and integrate the multiple components together. Test it.
- (2 marks) Create the full **fsm.vhd** and test it.

Sample Design and Testing Process

Develop your solution by breaking it down piecewise. Build a small, simple piece and test it. Continue building and testing each piece one-by-one as you go along (do not leave all testing until the end!).

1. Inside **blackjack.vhd**, connect one instance of **card7seg.vhd** to SW[3:0] and HEX0. Write the decoder in **card7seg.vhd**. Compile and test for all card values.

2. Create **datapath.vhd** and connect it inside **blackjack.vhd** to the **card7seg.vhd** components.
3. Create **dealcard.vhd** to continuously shuffle the deck and provide a new card every clock cycle on its output. Include it in **datapath.vhd** to deal one new card and display it on HEX0 every time you press KEY0. Repeating KEY0 presses should give a random card sequence.
4. Write a simplified **scorehand.vhd** by treating Ace/Jack/Queen/King as 10/11/12/13 (this is to keep things simple at first; you will fix it later). At first, connect it to sixteen switches SW[15:0] and verify that it works as expected. While testing, continuously display the player's hand on HEX[3:0], and the hand total on LEDR[4:0].
5. Modify **blackjack.vhd** to add the simple FSM. Create the simple **fsm.vhd**, having it build dealer and player hands of 2 cards each. Display both hands on 7-segment displays. Display the score of the player and dealer hands on LEDR[4:0] and LEDG[4:0]. Compile and test.
Completing the above is sufficient for week 1.
6. Modify **scorehand.vhd** to count Jack, Queen, and King as 10. If hand total exceeds 21, force hand total to be 31. Calculate "stand" and "bust" outputs; for testing this step only, temporarily display "stand" and "bust" on any green LED. Compile and test.
7. Modify **datapath.vhd** and/or **fsm.vhd** to use newPlayerCard[3:0] and newDealerCard[3:0] control signals. These signals indicate when each card should be sampled from **dealcard.vhd**
8. Modify **blackjack.vhd** and/or **fsm.vhd** to calculate and display the winner, reset the hands, and restart. Indicate the winner as required using LEDG. Compile and test.
9. Modify **blackjack.vhd** and/or **fsm.vhd**, checking "stand" and "bust" for player and dealer. The player's "stand" signal comes from SW0, the dealer's comes from **scorehand.vhd**. Compile and test.
10. Modify **scorehand.vhd** to count Ace as 11 or 1, as required.
11. Make sure all requirements are met.

Lab Demonstration

Demonstrate your BlackJack game to the TA. A golden solution will be provided so you can verify the instructions above – your solution should behave exactly the same.

VHDL Specifications

Use the top-level input/output specifications below. Do not add or remove any signals.

```

COMPONENT BLACKJACK IS
  PORT (
    CLOCK_50 : in std_logic;           -- A 50MHz clock
    SW       : in std_logic_vector(17 downto 0); -- SW(0) = player stands
    KEY      : in std_logic_vector(3 downto 0); -- KEY(3) reset, KEY(0) advance
    LEDR     : out std_logic_vector(17 downto 0); -- red LEDs: dealer wins
    LEDG     : out std_logic_vector(7 downto 0);  -- green LEDs: player wins

    HEX7 : out std_logic_vector(6 downto 0); -- dealer: fourth card
    HEX6 : out std_logic_vector(6 downto 0); -- dealer: third card
    HEX5 : out std_logic_vector(6 downto 0); -- dealer: second card
    HEX4 : out std_logic_vector(6 downto 0); -- dealer: first card

    HEX3 : out std_logic_vector(6 downto 0); -- player: fourth card
    HEX2 : out std_logic_vector(6 downto 0); -- player: third card
    HEX1 : out std_logic_vector(6 downto 0); -- player: second card
    HEX0 : out std_logic_vector(6 downto 0); -- player: first card
  );
END;

```

**DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
EECE 353 – Digital Systems Design**

Lab 4: VGA Controller / Drawing Lines

In this lab, you will be designing your own hardware to draw lines on the VGA screen. To get started, you will use the VGA Adapter Core from Lab 2. To add the line-drawing algorithm, you will get more experience creating datapaths and state machines than you did in Lab 3.

The top level diagram of your lab is identical to the one from Lab 2. The VGA Adapter Core is the part given to you, so all the excitement will be in the block labeled “your circuit”. Please review Lab 2.

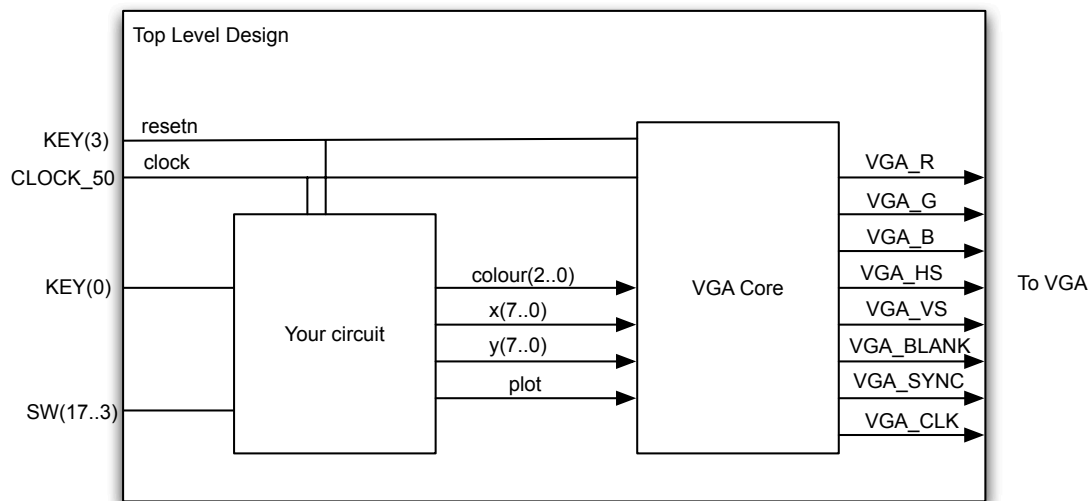


Figure 1: Overall block diagram. The top-level file is lab4vga.vhd.

Week 1: Simple Line Algorithm

The Simple Line algorithm draws a diagonal line across the screen. The basic algorithm is as follows

```
function line()
  dx := 0
  dy := 0
  loop
    setPixel(x0,y0)
    if x0 < 160 then
      x0 := x0 + dx
    end if
    if y0 < 120 then
      y0 := y0 + dy
    end if
  end loop
end function
```

In the algorithm above, the **setPixel()** step will visit a pixel and change its colour. The **setPixel()** function is only called exactly once per pixel on the line (no more, no less).

Week 2: Bresenham Line Algorithm

The Bresenham Line algorithm is a hardware (or software!) friendly algorithm to draw lines between arbitrary points on the screen. The basic algorithm is as follows (taken from Wikipedia):

```
function line(x0, y0, x1, y1)
  dx := abs(x1-x0)
  dy := abs(y1-y0)
  if x0 < x1 then sx := 1 else sx := -1
  if y0 < y1 then sy := 1 else sy := -1
  err := dx-dy
  loop
    setPixel(x0,y0)
    if x0 = x1 and y0 = y1 exit loop
    e2 := 2*err
    if e2 > -dy then
      err := err - dy
      x0 := x0 + sx
    end if
    if e2 < dx then
      err := err + dx
      y0 := y0 + sy
    end if
  end loop
end function
```

The algorithm is efficient: it contains no multiplication or division, and the **setPixel()** function is only called exactly once per pixel on the line (no more, no less).

Your circuit must behave as follows.

1. The switch KEY(3) is an asynchronous reset. When the machine is reset, it will start clearing the screen (hint: the circuit from Lab 2 can be used to clear the screen by changing the colour of each pixel to Black). Of course, clearing the screen will take at least 160*120 cycles.
2. WEEK1: Once the screen is cleared, initialize the <x0> and <y0> values to 0.
3. WEEK2: Once the screen is cleared, the user will set switches 17 down to 3 to indicate a point on the screen, and switches 2 down to 0 to indicate a colour. Specifically, SW(17 down to 10) will be used to encode the x coordinate of the point and SW(9 down to 3) will encode the y coordinate of the point. Finally, SW(2 down to 0) will indicate one of 8 possible colours used to draw the line.
4. When the user presses KEY(0), the circuit will draw a line. For WEEK 1, draw the diagonal. For WEEK 2, draw from the centre of the screen (location 80,60) to the position specified by the user. Of course, this will take multiple cycles; the number of cycles depends on the length of the line.
5. WEEK 2: Once the line is done, the circuit will go back to step 3, allowing the user to choose another point. Note that the screen is not cleared between iterations. At any time user can press KEY(3), the asynchronous reset, to go back to the start and clear the screen.

It is important to note that you are using CLOCK_50, the 50MHz clock, to clock your circuit.

Unlike Lab 3, we are not designing the datapath or FSM for you here.

You must clearly distinguish the datapath from the FSM in your VHDL code.

Challenge Circuit: (1 mark)

Challenge tasks are tasks that you should only perform if you have extra time, are keen, and want to show off a little bit. This challenge task is only worth 1 mark. If you don't demo the challenge task, the maximum score you can get on this lab is 9/10 (which is still an A+).

This challenge task is actually fairly easy.

- A. In the original circuit, you always connect the centre of the screen (80,60) to the point specified by the user. Modify your circuit such that, instead of starting from the centre, it starts from the point specified by the user in the previous iteration. So for the first iteration, if the user specifies point (x_0, y_0) , a line is drawn from the middle of the screen to (x_0, y_0) . Then, in the second iteration, if the user specifies point (x_1, y_1) , a line is drawn from (x_0, y_0) to (x_1, y_1) . In iteration i , a line is drawn from point (x_{i-1}, y_{i-1}) to (x_i, y_i) .
- B. (optional) Rather than taking the point from the switches, choose each point and colour pseudo-randomly (using a Linear Feedback Shift Register). Draw the lines quickly without waiting for KEY(0) between lines. Note that you will have to add some delay, or the screen will fill up too quickly for you to see.

Week 1 Grading: 2 marks (initializing screen black)
2 marks (structure of FSM and Datapath; clear separation of code)
1 mark (drawing diagonal)

Week 2 Grading: 2 marks (drawing lines works for typical/easy cases)
2 marks (drawing lines works for all cases)
1 mark (challenge circuit)