
# NAVAL POSTGRADUATE SCHOOL

**MONTEREY, CALIFORNIA**

# CAPSTONE

**EXPLORING CUSTOM HARDWARE TO ACCELERATE A PROGRAM BY LEVERAGING PARALLELISM**

by

Javen E. Davis

June 2021

Thesis Advisor: Theodore D. Huffmire
Second Reader: Vinnie Monaco

THIS PAGE INTENTIONALLY LEFT BLANK

| REPORT DOCUMENTATION PAGE | *Form Approved OMB No. 0704-0188* |
|---|---|



| **1. AGENCY USE ONLY** *(Leave blank)* | **2. REPORT DATE** June 2021 | **3. REPORT TYPE AND DATES COVERED** Capstone |
|---|---|---|

**4. TITLE AND SUBTITLE**
EXPLORING CUSTOM HARDWARE TO ACCELERATE A PROGRAM BY LEVERAGING PARALLELISM

**5. FUNDING NUMBERS**

**6. AUTHOR(S)** Javen E. Davis

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Naval Postgraduate School
Monterey, CA 93943-5000

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
N/A

**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES** The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**
Approved for public release. Distribution is unlimited.

**12b. DISTRIBUTION CODE**
A

**13. ABSTRACT (maximum 200 words)**

It is essential that computer science students learn how to leverage parallelism to accelerate their applications due to the dramatic slowing of sequential CPU performance. One way that computer scientists can leverage parallelism is through custom hardware like field-programmable gate arrays (FPGAs). Unfortunately, traditional FPGA development requires years of training in electrical engineering, digital design, and hardware description languages (HDLs), like Verilog or VHDL, and rigid electronic design automation (EDA) tools. To address this problem, this capstone developed a simple lab for a university-level computer architecture course that provides students with hands-on experience with custom hardware, allowing students with typical programming knowledge skills to learn how to use FPGAs without requiring a steep learning curve. Utilizing the Xilinx ecosystem, which consists of a low-cost FPGA board, the Vivado suite of design tools, and an intuitive programming language, this capstone project resulted in the successful implementation of a lab activity and manual involving the conversion of an adder circuit from Python to a working circuit on a Xilinx FPGA.

**14. SUBJECT TERMS**
custom hardware, field-programmable gate array, FPGA, parallelism, parallel computing, Xilinx University, Xilinx ecosystem, PYNQ-Z1, Vivado design suite, electronic design automation, EDA

**15. NUMBER OF PAGES**
63

**16. PRICE CODE**

| **17. SECURITY CLASSIFICATION OF REPORT** | **18. SECURITY CLASSIFICATION OF THIS PAGE** | **19. SECURITY CLASSIFICATION OF ABSTRACT** | **20. LIMITATION OF ABSTRACT** |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | UU |

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

**Approved for public release. Distribution is unlimited.**

# EXPLORING CUSTOM HARDWARE TO ACCELERATE A PROGRAM BY LEVERAGING PARALLELISM

Javen E. Davis
Lieutenant, United States Navy
BBA, Georgia State University, 2013

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL**
**June 2021**

Approved by: Theodore D. Huffmire
Advisor

Vinnie Monaco
Second Reader

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

# ABSTRACT

It is essential that computer science students learn how to leverage parallelism to accelerate their applications due to the dramatic slowing of sequential CPU performance. One way that computer scientists can leverage parallelism is through custom hardware like field-programmable gate arrays (FPGAs). Unfortunately, traditional FPGA development requires years of training in electrical engineering, digital design, and hardware description languages (HDLs), like Verilog or VHDL, and rigid electronic design automation (EDA) tools. To address this problem, this capstone developed a simple lab for a university-level computer architecture course that provides students with hands-on experience with custom hardware, allowing students with typical programming knowledge skills to learn how to use FPGAs without requiring a steep learning curve. Utilizing the Xilinx ecosystem, which consists of a low-cost FPGA board, the Vivado suite of design tools, and an intuitive programming language, this capstone project resulted in the successful implementation of a lab activity and manual involving the conversion of an adder circuit from Python to a working circuit on a Xilinx FPGA.

THIS PAGE INTENTIONALLY LEFT BLANK

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF ACRONYMS AND ABBREVIATIONS

| | |
|---|---|
| ACP | accelerator coherency port |
| ALU | arithmetic logic unit |
| ASIC | application-specific integrated circuit |
| BRAM | block RAM |
| CLB | configurable logic blocks |
| CPU | central processing unit |
| EDA | electronic design automation |
| FF | flip flops |
| FPGA | field programmable gate arrays |
| GPU | graphics processing units |
| HDL | hardware description languages |
| HLS | high-level synthesis |
| IP | intellectual property |
| IPI | intellectual property integration |
| LUT | look-up tables |
| PL | programmable logic |
| PS | processing system |
| PYNQ | Python productivity for Zynq |
| RAM | random-access memory |
| RTL | register transfer level |
| SoC | system on a chip |
| VHDL | very high-speed integrated circuit hardware description language |
| WSC | warehouse-scale computers |

THIS PAGE INTENTIONALLY LEFT BLANK

# ACKNOWLEDGMENTS

Foremost, I must thank my better half, Consuela, for always being there to provide exactly what I need, whether it be a listening ear or a pep talk. I would not have been able to cross the finish line without you.

A special thanks is offered to my cohort, Nick Carter and Pete Pommer, for the camaraderie, countless hours, and memories that I will cherish.

I also want to thank Ben Travers for allowing me to join your thesis idea. It has been a pleasure working with you on the project, even though we had to split thesis work.

Lastly, I want to thank my advisors, Dr. Huffmire and Dr. Monaco, for all their help along the way.

THIS PAGE INTENTIONALLY LEFT BLANK

# I. INTRODUCTION

This chapter is an overview of the thesis that explains the reasoning behind the research and outlines the expectations.

## A. RESEARCH PURPOSE

Custom hardware is a foundational topic in modern computer architecture courses at both undergraduate and graduate levels. Hands-on experience with real hardware is essential to learning how to leverage custom hardware to accelerate applications. The reason for using custom hardware is that sequential performance of central processing units (CPU) is not increasing, and therefore, computer scientists have no other choice than to leverage parallelism in order to accelerate their applications. Custom hardware is capable of accelerating high-throughput applications by leveraging the parallelism of billions of transistors provided by Moore's Law. This can be accomplished via a host of options like graphics processing units (GPUs), warehouse-scale computers (WSCs), multi-core CPUs, application-specific integrated circuits (ASICs), or field programmable gate arrays (FPGAs). Different kinds of machines leverage different kinds of parallelism. This thesis will focus on FPGAs. Traditional FPGA development requires years of electrical engineering training with extensive knowledge of hardware description languages (HDL) or very high-speed integrated circuit hardware description language (VHDL) and difficult lab environments with rigid tools that only work on Windows platforms. The Xilinx University ecosystem consists of the PYNQ board (PYNQ stands for Python productivity for Zynq), Vivado software suite of design tools which are free, and a programming language that CS students are already familiar with. The PYNQ board claims that it is a low-cost alternative to traditional FPGAs. This thesis will test out that claim by creating a simple lab that will enable students to learn to use FPGAs sans the steep learning curve.

## B. RESEARCH OBJECTIVE

The main objective is to analyze and explore the utilization of the Xilinx ecosystem, PYNQ-Z1 FPGA, along with the Vivado design software suite to leverage parallelism to accelerate a program.

## C. RESEARCH QUESTION

How does the technology of the Xilinx ecosystem, composed of the PYNQ board along with Vivado design software suite, allow a computer science student to leverage parallelism to accelerate a program, in lieu of any electrical engineering training?

## D. SCOPE OF THESIS

This thesis will explore custom hardware with an emphasis on FPGAs and how they are used to accelerate a program. The central focus of this study is the Xilinx ecosystem. I will conduct a thorough analysis of the PYNQ-Z1 FPGA and the Vivado design tools, which will be the only hardware used in this thesis. Python will be the only language used to program, and the code will be developed and tested directly on the PYNQ board. This thesis will not explore any of the other custom hardware options nor will any other FPGA, design tools, or programming language be used.

## E. BENEFITS OF STUDY

The key benefit of this research will be allowing students to gain hands-on experience with real hardware via low-cost FPGAs and free design tools that are easy to use. This will create a pathway for future CS students to develop familiarity with and begin working with the Xilinx ecosystem.

## F. THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

- Chapter II: Background

Chapter II details the history and current state of custom hardware. It examines parallel computing via CPUs, GPUs, FPGAs, ASICs, and WSCs while emphasizing the importance of the FPGA regarding economic and performance tradeoffs.

- Chapter III: Xilinx Ecosystem

Chapter III discusses each component of the Xilinx ecosystem, how a program converted into a circuit, and how parallelism is leveraged to accelerate a program.

- Chapter IV: Lab Manual

Chapter IV describes in great detail the technical implementation of the Xilinx ecosystem to leverage parallelism to accelerate high-throughput computation sans the steep learning curve of traditional design methods.

- Chapter V: Conclusion and Future Work

Chapter V summarizes the research conducted for this thesis. Finally, recommendations are provided for future work to expand the usability of the Xilinx ecosystem.

THIS PAGE INTENTIONALLY LEFT BLANK

# II. BACKGROUND

This chapter explores parallel computing via custom hardware with emphasis on the FPGA and its importance.

## A. INTRODUCTION

According to Ross, Gordon Moore predicted that the number of transistors on a chip would double annually in 1965 [15]. This prediction is now widely recognized as Moore's Law. Moore's Law resulted from an observation of the trends in the fabrication of transistors, which was likely to continue for the next ten or more years, increasing chip transistor counts to above 65000 [15]. The prediction remained true for the next two decades as architectures exploited the increase in transistor speed and huge transistor budgets from silicon technology to double performance roughly every 18 months. Figure 1 shows the trends as predicted by Moore's Law:

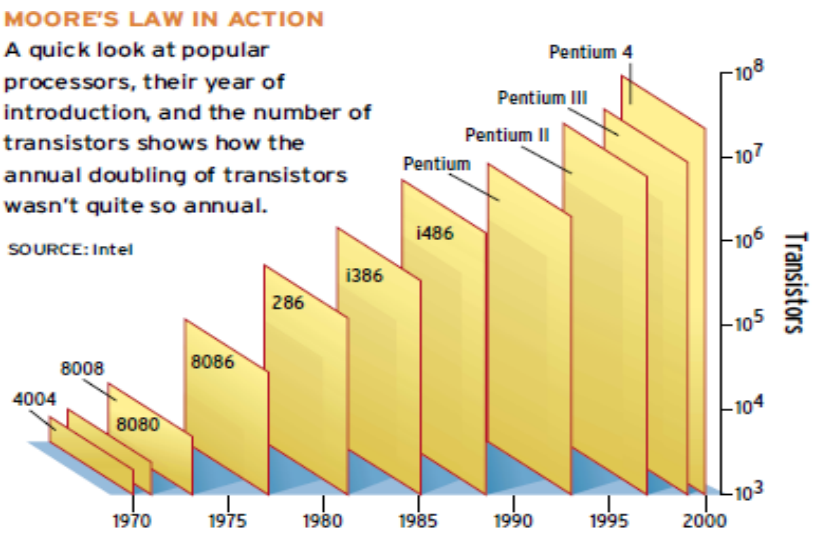


Figure 1. Moore's Law in action. Source: [15].

Having more transistors on a chip does not necessarily result in efficient computation unless a CPU design can utilize them successfully. Moore's Law has advanced to the point that billions of transistors can fit on a single chip. How to use billions

of transistors in an efficient design is a crucial issue for computer architecture. Modern multi-core processors are known as chip multi-processors, where many processors reside on the same chip [11]. Due to a multitude of technical challenges, the sequential performance of uni-core CPUs is no longer increasing, and computer scientists have no other choice than to leverage parallelism to accelerate their applications. Custom hardware can accelerate high-throughput applications by leveraging the parallelism of billions of transistors provided by Moore's Law. This can be accomplished via a host of options like GPUs, WSCs, multi-core CPUs, ASICs, or FPGAs. Different kinds of machines leverage different types of parallelism [16]. This thesis will focus on how FPGAs are used in parallel computing to accelerate high-throughput computation.

## B. PARALLELISM AND THE POWER WALL

The performance of the central processing unit is no longer improving exponentially. It is unlikely to handle high processing demands due to the rapid increase in the amount of data. The inability to increase the CPU speed because of the rise in power demands is called the 'power wall.' The power wall limited the rate of advancement in single-core processors. Computer architects decided to use several processors in one chip for power efficiency vice single processors, which would gradually increase in number in relation to Moore's Law. Therefore, with each computing generation, the number of processors or cores was doubled in the multi-core processor. The multi-core processors utilize a single chip, are power-efficient, and have a high clock rate. These multi-core processors enable parallel computing via thread-level parallelism.

Parallel computing utilizes the increasing number of microprocessors on a chip via thread-level parallelism. However, the increase in microprocessors has not eliminated the power wall. Accessing dynamic random-access memory, a computer's main memory still requires far more cycles than performing a single basic arithmetic logic unit (ALU) operation [16]. This is known as the memory wall, as communication cost dominates computation cost. The switch to multi-core poses new challenges, in particular with programmability, bandwidth, and consistency.

However, not all workloads are likely to be speedily executed while operating in parallel because some algorithms are hard to parallelize due to sequential bottlenecks, dependencies, high communication cost, and synchronization/consistency issues. Although parallelism is not a panacea, it is the only way forward; computer scientists have no other way to accelerate their applications, as faster single-threaded, sequential, uni-core CPUs will not be forthcoming due to the issues discussed above. Therefore, future computer scientists must be aware of the different kinds of parallelism (e.g., thread-level, instruction-level, data-level, request-level, etc.) exploited by different kinds of processors (e.g., CPUs, ASICs, FPGAs, GPUs, SoCs, etc.). The best way to attain top speeds in hardware is through leveraging parallelism via the innovation of accelerators on FPGAs, ASICs, GPUs, WSCs, and multi-core CPUs. These accelerators increase the performance of high-throughput applications by leveraging the parallelism of billions of transistors. Accelerators are already in use in data centers such as Google and Microsoft. They are also in use in other domains such as graph analytics, machine learning, and genome sequencing.

## C. SUMMARY OF THE ACCELERATORS

Accelerators are implemented in custom hardware because they can leverage a multitude of transistors to perform logic functions in parallel. The use of custom hardware circuits, commonly known as application-specific integrated circuits, enables a task to attain precisely the needed functionality as quickly as possible. ASICs are also small, cheap, and faster chips designed to consume as little power as possible. ASICs enable a computer to function orders of magnitude faster while using orders of magnitude less power than a general-purpose CPU for high-throughput workloads [1]. The main drawback of using an ASIC chip is that it only performs the function for which it was designed for. If the function is slightly modified, the chip might not function and need modifications, which will be expensive due to the challenges of chip design and fabrication. The GPU, in contrast to the ASIC, is far more flexible, as it can be programmed using a modified version of the C programming language and a special compiler. GPUs can be used for graphics, of course, but can also be used for general-purpose, high-throughput computing tasks.

Custom accelerators can also be implemented using field program gate arrays, also known as programmable hardware. FPGAs combine the programmability of a CPU with the performance of an ASIC. A CPU is a jack-of-all-trades, while an ASIC is a master of one trade. On a continuum between flexibility and performance, CPUs are the most flexible, followed by GPUS, then FPGAs, and finally ASICs. A FPGA can be attached to a CPU as a coprocessor. FPGAs can accelerate a variety of throughput computing workloads, image processing, data mining, database operations, bioinformatics, n-body simulations, K- means, and AES encryptions [1]. This thesis will analyze FPGAs in detail in the subsequent chapters.

## D. THE FPGAS

A field-programmable gate array can be configured to implement virtually any digital circuit. FPGAs combine the programmability of CPUs with the performance of custom hardware. FPGA applications are programmed using HDLs such as Verilog and VHDL. Electronic design automation (EDA) software such as the Xilinx Vivado software suite translates the HDL to a bitstream, which is loaded onto the FPGA and is used to configure the FPGA. FPGAs can also be programmed using high-level languages via a methodology known as electronic system-level design, in which a high-level representation (e.g., SystemC, HandelC, MATLAB, etc.) is translated to HDL and then to a bitstream. An FPGA consists of an array of configurable logic blocks (CLBs), each of which implements an individual, primitive logic gate (e.g., AND, OR, XOR, etc.). The CLBs are islands that reside in a sea of configurable interconnect, which connects the primitive logic gates together to form higher-level functionality (e.g., half adders, full adders, ripple-carry adders, etc.). The bitstream consists of the bits that configure the CLBs along with the bits that configure the interconnect.

In combinational circuits, primitive logic gates such as AND, XOR, NOR, OR, and NAND can be composed to implement higher-level functionality, from half adders to full adders to ripple-carry adders to ALUs to CPUs to full systems. In ASICs, the logic gates cannot be modified after fabrication [1]. If a bug is discovered in a custom circuit, with a FPGA, the HDL can be modified, and a new bitstream can be generated, much like software

patch for a CPU. This is not the case with Application Specific Integrated Circuit (ASICs) [5]. Since an ASIC is fixed after fabrication, a flaw discovered after manufacturing cannot be fixed; therefore, ASIC design requires rigorous and costly verification to ensure that the design is free of bugs prior to tape-out.

FPGAs have improved over time with respect to their performance, logic cell capacity, and reconfiguration time. The early types of FPGAs were small devices only capable of prototyping a small circuit or sub-circuit, and they needed a few seconds to be configured. Even these early, primitive FPGAs made it possible for engineers to test prototype candidate circuits in order to efficiently explore the design space. The configurability of FPGAs also provides the ability to upgrade hardware far more efficiently than ASICs. Newer FPGAs are now taking milliseconds to reconfigure, and it is estimated that soon they will be taking only 100 microseconds only to reconfigure. This makes FPGAs ideal for complex applications requiring rapid adaptations [8].

A FPGA has thousands of CLBs that reside within its numerous programmable interconnects. Each CLB implements a primitive digital logic gate as a look-up in its corresponding truth table. Therefore, the FPGA is unique from the CPU and the GPU because it can be configured to be virtually any digital circuit, including a CPU. When a bitstream implements a CPU, it is called a soft IP core (IP stands for Intellectual Property). Examples of soft CPU cores include Nios II, and Xilinx MicroBlaze. Many FPGAs also have hard-wired circuits like CPUs (e.g., PowerPC) and memory blocks that reside alongside the reconfigurable fabric. A hard-wired CPU that is built into the FPGA is called a hard IP core. Since an FPGA can be configured to implement an Ethernet controller, FPGAs have an advantage over GPUs because they can communicate directly with the network interface, bypassing intermediate bus interfaces like USB or PCI, as is the case in other accelerators such as GPU. However, FPGA design is more complex and challenging than other architectures such as GPU or CPU and requires a higher learning curve to understand and use [5] because while programming a GPU or CPU requires only knowledge of a C-like programming language, FPGA development requires electrical engineering training and knowledge of unwieldy hardware description languages like Verilog and VHDL. Therefore, this means that it is, in most cases, used to solve niche

problems. This thesis aims to investigate whether it is feasible for computer scientists, who may lack years of formal electrical engineering training and FPGA design experience, to use FPGAs to accelerate their applications with their mere knowledge of C-like programming languages.

The ability of FPGAs to adapt to new tasks after being initially programmed makes them suitable for applications such as artificial intelligence and neural networks. The high throughput of FPGAs makes them useful in real-time modeling as well. Some FPGAs are capable of dynamic partial reconfiguration, allowing them to adapt to changing workloads over time. FPGAs offer ample parallelism and unique economic tradeoffs with respect to ASICs. For low-volume projects, FPGAs are more economical than ASICs. However, if there is sufficient demand for a chip, the cost of ASIC development can be amortized over a large number of customers. Crossover volume is the volume at which the transition from FPGA to ASIC becomes economically viable. FPGAs are slower than ASICs at the same technology node (i.e., feature size), but the gap between ASIC and FPGA narrows if economic constraints force the ASIC to be fabricated at an older technology node to save cost [9]. FPGAs have become so powerful that they can be used to implement systems (e.g., the Mars Rover) rather than merely prototype them. Fewer and fewer companies can afford the high non-recurring engineering cost of ASIC fabrication, especially for low-volume projects that are common in defense, national security, and high-assurance applications. FPGAs are commonly used in warehouse-scale computing to accelerate web searches [12] and voice recognition for cloud computing and artificial intelligence. Therefore, the general public uses FPGAs on a daily basis without even knowing it when performing an everyday task as mundane as a web search.

## E. FPGA ARCHITECTURE

FPGAs have evolved from small devices used for prototyping small circuits or sub-circuits to huge arrays of programmable logic and interconnects having on-chip memories, microprocessors, and other custom data paths. Modern FPGAs are highly capable devices with ample logic cell capacity to prototype a multi-core CPU. To leverage the capabilities of FPGAs in a more accessible fashion than traditional computer engineering methods,

programmers are designing digital circuits with EDA tools to automatically translate the abstract specifications to physical layout [6]. High-level synthesis (HLS) design flows translate a high-level specification of an algorithm to register transfer level (RTL) and then to a bitstream. The FPGA logic blocks are often implemented as look-up tables (LUT) shown in Figure 2.

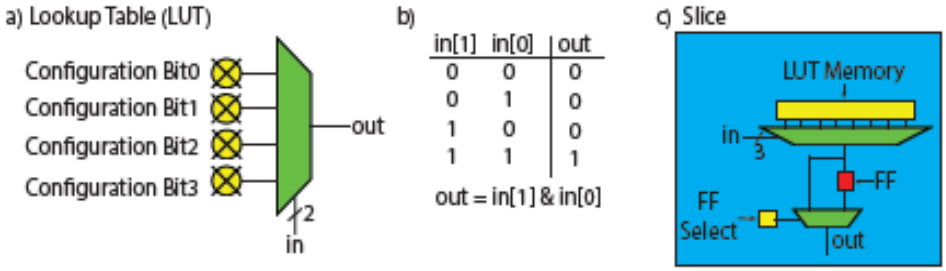


Figure 2. FPGA input and output look-up tables. Source: [8]

The LUT in part A has four configuration bits that correspond to the output column of a truth table of the desired logic gate. The table in part b shows a sample truth table for the AND logic gate with values matching the configuration bits. In part C of the above figure is a simple slice (a combination of LUTs, Multiplexers, and Flip Flops) with a somewhat complex LUT with eight configuration bits corresponding to a three-input logic gate. In practice, however, extensive research has proven that the optimal size of a LUT corresponds to a small logic gate (e.g., a 2-input gate) [14].

Static RAM FPGAs use Flip Flops (FF) as their basic memory cells and collocate them with LUTs. In some instances, however, LUTs and GAs can be combined with other specialized functions leading to the formation of complex logic elements known as a CLB, slice, or Logic Array Block depending on the preferences of the vendor [7]. A FPGA also has a programmable interconnect that provides a connection between slices. As the number of slices increase, their input and outputs get connected to a routing channel, which has a lot of configuration bits that determine if the inputs and outputs get connected or not. The routing channels are further connected to switch boxes that act as pass transistors, enabling

the routing tracks' connection from one routing channel to the next [5]. Figure 3 shows how the slice, switch box, and routing channel in an FPGA are interconnected.

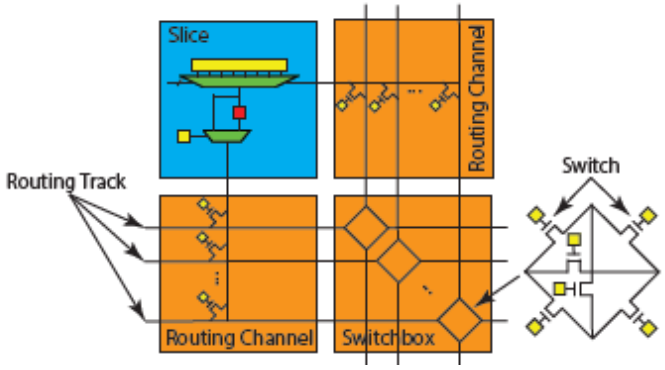


Figure 3. The interconnection of the routing channel, slice, and switch box in an FPGA. Source: [8].

Typically, the FPGA is designed to have a 2D representation. It is designed to provide 2D abstraction during computation in what is known as 'Island-style' architecture. That simply means that the slices are like 'logic islands' connected using switch boxes and routing channels [5]. The 2D dimension of the FPGA is shown in Figure 4.

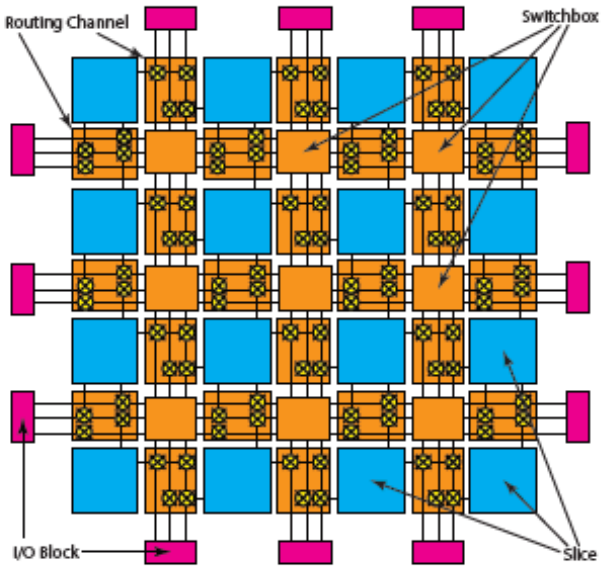


Figure 4. 2D structure of the FPGA with an Island-style structure. Source: [8]

As the FPGA gets bigger due to a continuous increase in transistors provided by Moore's Law, the architectures have started to use "hard" resources designed to perform specific functions [5]. A good example of such a resource is the DSP 48 Datapath designed to perform mathematical operations such as multiplication and addition and other word-level logical operations for digital signal processing. A modern FPGA, therefore, may have thousands of DSP 48 Datapath across its logical fabric [8].

Other hardened resources are Block RAM (BRAM) and on-chip microprocessors. BRAM is a configurable random-access memory module that can support many memory layouts and interfaces and allow the storage of large data sets. BRAM enables the FPGA to leverage parallelism by being used as configurable register files that can communicate with the DSP48 Datapath and on-chip microprocessors while at the same time transferring data [5]. A FPGA can include anywhere between one and four hard-wired microprocessors, with the largest FPGA having four on-chip microprocessors. Microprocessors are necessary because they offer control of the system and can efficiently run software. Microprocessors also initiate data movement between the sensors, memory, and controller of the system [1].

## F. FPGA FEATURES FOR PARALLELISM

FPGAs leverage parallelism by implementing custom circuits. The type of parallelism depends on the bitstream [16]. The ability to customize the FPGA enables it to be very fast because it executes specific tasks that efficiently utilize all the resources. For example, a CPU is designed to perform generic functions; therefore, tasks do not utilize all the resources. Moreover, the ability of the FPGA to be customizable leads to low power consumption as compared to when using CPU or GPU. A FPGA is also able to support the processing of data at low latency because of its design architecture, which does not have any specific instructions [1]. The task can also be fast because the results are moved to the next component for processing instead of sending it to the memory before it gets relayed to the required components. While FPGAs typically operate at much lower clock frequencies than CPUs, FPGAs can outperform CPUs on high-throughput workloads by orders of magnitude with respect to power and performance. However, in general, CPUs are more suitable than FPGAs for applications requiring low latency. Fortunately, latency is often not a priority for high- throughput workloads.

## G. FPGA DESIGNS AND PROGRAMMING FOR PARALLELISM

The design space of FPGAs and bitstreams is vast. However, the common FPGA designs have a large number of logic blocks that are easily configured and a programmable grid of connections that the designer uses to connect the blocks to form any design that they desire. FPGA bitstreams can either be coarse-grained or fine-grained. The coarse-grained bitstreams have a small number of IP blocks, whereas the fine-grained structures have many simple blocks as small as single logic gates. Therefore, designers can choose to create either a course-grained bitstream or a fine-grained bitstream depending on what application they want to implement and the time they have to build the system and subsystems. However, for adaptive workloads, dynamic designs capable of dynamic partial reconfiguration (i.e., swapping) are recommended, although engineering a “hot-swappable” system that uses partial dynamic reconfiguration is very challenging, with many engineers avoiding it entirely except for a few proven applications, e.g., a crossbar

switch. Such a design would allow the chip to operate in either time-sharing mode or be utilized in other successive configurations [5].

Programming the user-defined logic in a FPGA uses hardware description languages (HDL) such as Verilog and VHDL. HDLs are distinct from the programming languages because HDLs describe and define digital circuits rather than software code. Therefore, HDLs reflect the parallel nature of hardware. Unlike a programming language like C or Java, with subroutines containing code that is executed sequentially, HDL files can describe circuits containing many logic gates or IP blocks that operate in parallel. However, HDL design, in contrast to C programming, requires advanced skills in other fields such as digital electronics design, the running of components in parallel, and task and resource trade-offs, among others. However, HLS tools allow mere programmers to implement their applications using standard languages such as C or C++. Compared to ASICs, FPGA solutions do not require fabricating a chip, although both require a working digital circuit to be designed in HDL [5]. The same digital design can either target an ASIC, which requires costly fabrication in a foundry, or an FPGA, which can be programmed by merely connecting the FPGA board to a PC workstation. In the case of ASIC design, the HDL goes through one set of tools to create a geometrical database standard for information interchange file that is sent to the foundry; in the case of FPGA design, the same HDL is sent through a different set of tools that generate a bitstream that is used to configure the FPGA device. In other words, the same HDL can either target an ASIC or an FPGA, depending on the requirements (and budget) of the customer.

## H. DEPLOYING FPGA

There are many ways to deploy a FPGA depending on where designers are trying to ensure there is an acceleration of the speed of execution, and the task is done. If trying to accelerate a database, for instance, designers can deploy the FPGA between the data source and the CPU, where it will be used as a filter, or they can deploy it as a co-processor where it is used to speed to workload of loading tasks. The other commonly used deployment method is attaching a FPGA as an input-output attachment [10].

Attaching a FPGA as an input-output accelerator is mostly used in situations where the CPU encounters a bottleneck. Therefore, the FPGA relieves the bottleneck by executing tasks offloaded from the CPU. In such cases, the FPGA can be connected using buses like PCIe, and it operates using independent memory from the CPU. Similar to a GPU-CPU coprocessor arrangement, when tasks get offloaded to the FPGA, the system copies the host (CPU) memory task to its device (FPGA) memory, where it gets queued for processing. The task is then executed, and the system then copies the result back to the host memory. The I/O mechanism is often an FPGA card (i.e., a circuit board containing the FPGA coprocessor with a PCI interface to the motherboard) and is used mainly when coprocessing high-throughput applications [10].

In an alternative CPU-FPGA coprocessor arrangement, the FPGA can directly access memory from the CPU using shared memory. The shared memory is often larger than the device memory, and therefore in comparison with the I/O attached deployment, it can access data in the shared memory. A FPGA can be deployed as a co-processor using two methods. The first way is through tightly coupling the FPGA and CPU together in the same die of the socket. Examples of connecting the FPGA and CPU include the Intel Xeon + FPGA platform. In the Intel Xeon + FPGA platform, the CPU and FPGA get connected through Intel QuickPath Interconnect as well as Accelerator Coherency Port *(*ACP*)* [5]. In the second method, the FPGA gets connected to the host device as a co-processor using coherent I/O interfaces. Some of the coherent I/O interfaces are IBM Coherent Accelerator Processor Interface, enabling the FPGA to access extra bandwidth from the host [5]. However, with each passing day, more new ways of deploying the FPGA are being devised. Most of the new methods refine the already existing methods that aim to ensure data transfer is as quick as possible.

The FPGA functions efficiently in in-line processing, whereby functions can be subdivided and work on a stream without each process, knowing the data being processed in other processes. Examples of in-line processing are matrix filters and dilation. In these processes, data can be processed in small separate sections making it suitable for the processing of high data rates. For these reasons, the FPGA is very good for use in the processing of images where a lot of data needs to be processed in a very short time [5].

However, the FPGA is not good for use in processing the whole image because if an image has a lot of data, and with the FPGA having a small memory, it cannot manage to store the whole image. It is also not good for use in iterative functions. The FPGA can only manage these functions if extra hardware is provided to assist in creating a virtual CPU.

## I. CHAPTER SYNOPSIS

This chapter conveyed that modern computer scientists wishing to accelerate their applications have no choice other than to embrace parallel computing via CPUs, GPUs, FPGAs, ASICs, and WSCs. It emphasized the importance of FPGA in this parallel computing landscape, as it offers a unique set of performance and economic tradeoffs with respect to its competitors. The FPGA allows some high-throughput applications to achieve orders of magnitude performance advantage and power savings over CPUs, taking the future of parallel computing to a whole new level.

The next chapter will cover the Xilinx Ecosystem.

THIS PAGE INTENTIONALLY LEFT BLANK

# III. XILINX ECOSYSTEM

This chapter gives an in-depth description of each component of the Xilinx University ecosystem and how they leverage parallel computing.

## A. XILINX COMPONENTS

The Xilinx ecosystem consists of the PYNQ board, the Vivado suite of design tools, and an intuitive programming language, Python.

## B. PYNQ OVERVIEW

Xilinx states that PYNQ, which is an acronym for Python Productivity for ZYNQ, is "an open-source project from Xilinx that makes it easier to use Xilinx platforms" [17]. The PYNQ board is a low-cost alternative to traditional FPGAs that utilizes free design tools and a well-known, widely used programming language. [13]

## C. PYNQ CAPABILITIES

One benefit of PYNQ is it can support many accelerator cards in a single server. PYNQ is also beneficial because, when enabled, it can be coded in Jupyter Notebook through the use of Python, a familiar programming language for Computer Scientists at any level. According to Xilinx, "PYNQ can be delivered in two ways; as a bootable Linux image for a Zynq board, which includes the pynq Python package, and as open-source packages or as a Python package for an Alveo or AWS-F1 host computer" [13].

## D. PYNQ OVERLAYS

The best way to explain how PYNQ works is through its overlays. Xilinx Zynq is an all-programmable device based on the core ARM Cortex-A9 processor, which is sometimes called the Processing System or PS amalgamated with FPGA fabric or Programmable Logic or PL [13]. For instance, image processing is a specific application that FPGAs can accelerate [13]. A software developer uses an overlay like a software library in implementing some of the functions for image processing like thresholding, edge detection, and many others on the FPGA fabric.

Software developers do not work at the signal but the interface level of abstraction when creating connections between IP modules, thus increasing productivity. In most cases, this involves using AX14, a standard interface in the industry, but the IP integrator also supports many other interfaces [13]. When designer teams are working at the interface level, they can swiftly compose complex systems that leverage IP modules. By leveraging HLS and Vivado IP Integration (IPI), customers can save development costs of up to 15X versus the RTL approach [13].

## E. VIVADO DESIGN SUITE TOOLS

Vivado design suite are free tools that according to Xilinx "delivers a SoC-strength, IP-centric and system-centric, next generation development environment that has been built from the ground up to address the productivity bottlenecks in system-level integration and implementation" [17]. Xilinx Vivado is also helpful because it does acceleration verification with the Vivado Rapid Verification Solution's assistance. Xilinx Vivado Design Suite's Simulator supports both timing and functional simulation [17].

## F. TURNING A PYTHON PROGRAM INTO A CIRCUIT

PYNQ enables the use of Zynq devices easily and efficiently [13]. The Xilinx design flow translates Python programs into circuits that run on the PYNQ platform [4]. Logic circuits are imported as libraries using APIs just like software libraries. The Xilinx can translate a Python program into a circuit if programmable logic circuits are presented in overlays or hardware libraries [13].

The overlays are analogous to software libraries. Software developers may decide to select overlays that best match their applications, and the overlays are accessible through APIs or application programming interfaces [13]. To create a new overlay will require experienced engineers to build or design programmable logic circuits. However, the significant difference with traditional methods is the paradigm of "build once" and "reuse severally." Therefore, a circuit can be developed through overlays like software libraries, which are meant to be configured and reused as many times as possible with different applications [13].

## G. LEVERAGING PARALLELISM TO ACCELERATE A PROGRAM

To understand how Xilinx and PYNQ leverage parallelism to accelerate a program/code, one needs to understand hardware libraries or overlays. Overlays can accelerate software. For instance, image processing is an example of an application that can benefit from the acceleration provided by custom hardware. Therefore, a software engineer can use overlays in the same manner as a library program to implement image processing functions like edge detection [13].

Hardware can be loaded dynamically to the FPGA as expected, just like software libraries. For instance, the implementation of separate image processing functions can be done in many different overlays then combined using Python [13]. PYNQ is in charge of providing a Python interface that will allow the PL overlays to be authorized from the PS running Python. This is easier than traditional FPGA design, which requires hardware engineering expertise and knowledge [13].

According to [13], hardware designers create PYNQ overlays, which are then combined using the PYNQ Python API. Therefore, software engineers can apply Python interfaces to control the hardware overlays without actually having to design the overlays themselves, thus accelerating the development time [13]. This is analogous to software libraries created by experienced developers which are then used by other software engineers who build applications. By leveraging HLS and Vivado IPI, clients can save up to 15X on development costs versus the RTL approach [13].

## H. CHAPTER SYNOPSIS

This chapter analyzed the components of the Xilinx ecosystem. It also covered how overlays are used to translate a program into a circuit and accelerate software.

In the next chapter, the technical implementation of the Xilinx ecosystem will be detailed.

THIS PAGE INTENTIONALLY LEFT BLANK

# IV. LAB MANUAL

This chapter illustrates the lab activity involving the technical implementation of leveraging parallelism to accelerate a program via the Xilinx ecosystem.

## A. OVERVIEW

The Xilinx ecosystem is a low-cost alternative to gain hands-on experience with FPGAs without the technical background or steep learning curve of traditional development methods. It consists of the PYNQ-Z1 FPGA, free Vivado design software suite, and the Python programming language. This lab will analyze and explore the utilization of the Xilinx ecosystem to leverage parallelism to accelerate a program.

## B. PYNQ-Z1 BOARD SET UP

Follow along with the guide or video to ensure the board is properly configured. The guide can be found at the following address:

< https://pynq.readthedocs.io/en/v2.6.1/getting_started/pynq_z1_setup.html> [13]. The video can be found at the following address:

< https://www.youtube.com/watch?v=SuXkbcK3w9E> [13]. Figure 5 shows the board after it was unboxed, and Figure 6 shows the board after configuration is complete. If configuration is successful, the board homepage can be accessed. Refer to Figure 7.

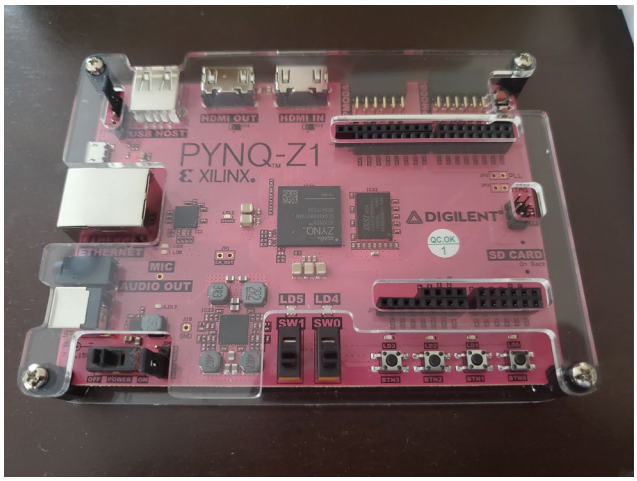


Figure 5. PYNQ-Z1 board prior to configuration

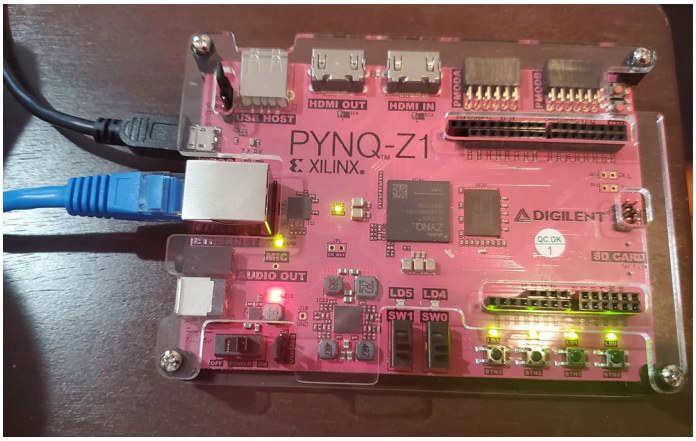


Figure 6. Fully operational PYNQ-Z1 board

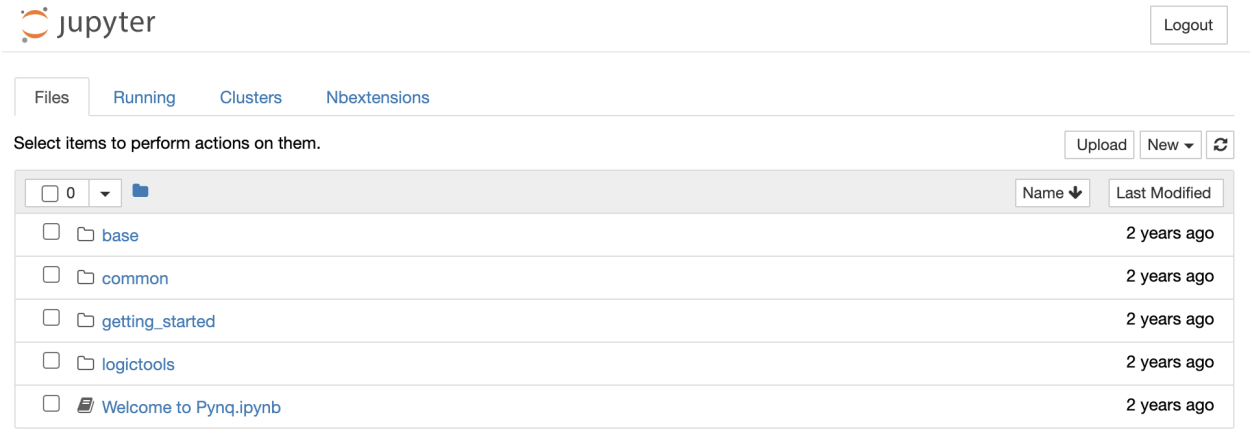


Figure 7. PYNQ-Z1 board homepage.

## C. DOWNLOAD VIVADO

Use the following link and steps to download Vivado:

<https://www.xilinx.com/support/download.html> [17] (Note: The files are extremely large so ensure you have ample space and time to complete the download.)

1. Select the correct version for your operating system. Refer to Figure 8.

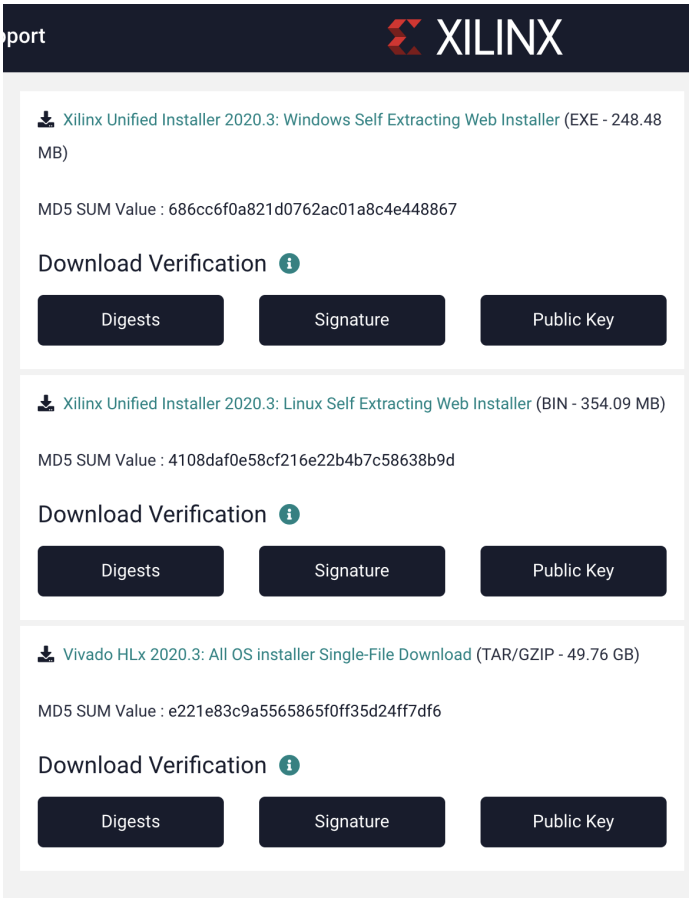


Figure 8. Xilinx download page. Source: [17]

2. Create a free account. (Note: Using your NPS email is recommended.)
3. Fill out the personal information in the download center then click "Download". Refer to Figure 9.

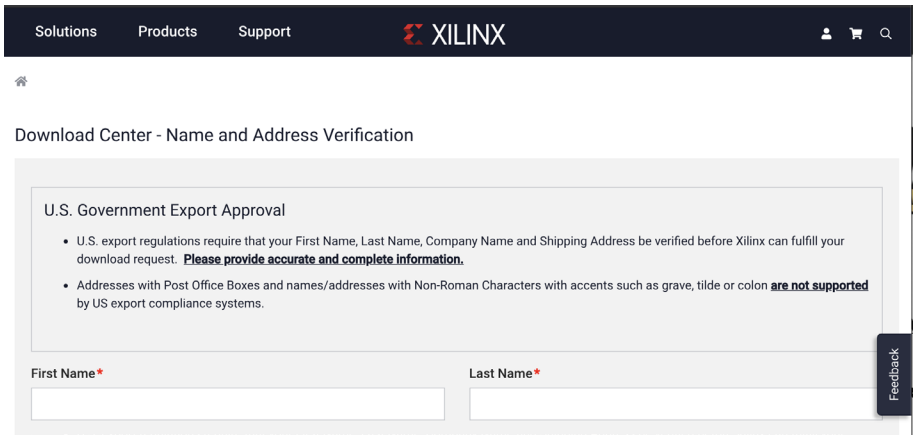


Figure 9. Xilinx download center. Source: [17]

4. Unzip the files.

5. Run "xsetup" to install the files. Refer to Figure 10. (Note: A Windows environment is needed to complete so Ubuntu is recommended if not already in use.)

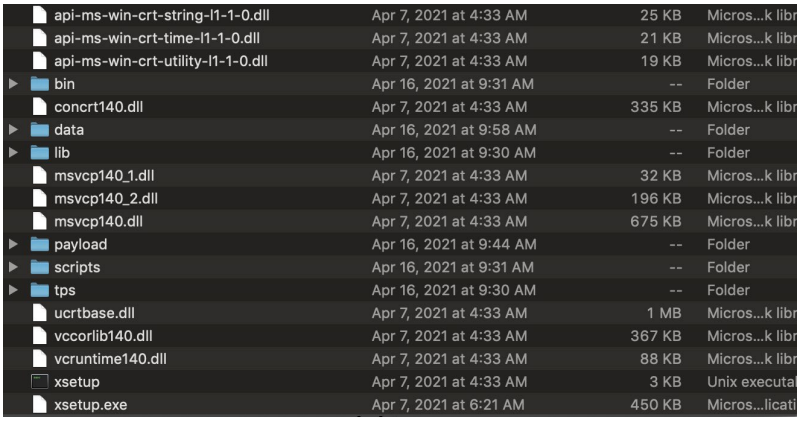


Figure 10. Unzipped Xilinx files.

## D. DOWNLOAD PYNQ-Z1 BOARD FILES

Use the following link to download the board files for the PYNQ-Z1 FPGA:

<https://pynq.readthedocs.io/en/v2.6.1/overlay_design_methodology/board_settings.html> [13]

## E. GENERATE AND IMPLEMENT THE BITSTREAM

Complete the following steps to "add two 32-bit integers together using a design with a single IP that was developed using HLS." [13]. The overlay tutorial can be found at the following link:

<https://pynq.readthedocs.io/en/v2.6.1/overlay_design_methodology/overlay_tutorial.html> [13]

Alternatively, you can follow along with this video:

<https://www.youtube.com/watch?v=Dupyek4NUoI> [18]

1. A. Open the Vivado HLS software then click "Create New Project" on the welcome page. Refer to Figure 11 and Figure 12.


Figure 11. Vivado HLS icon

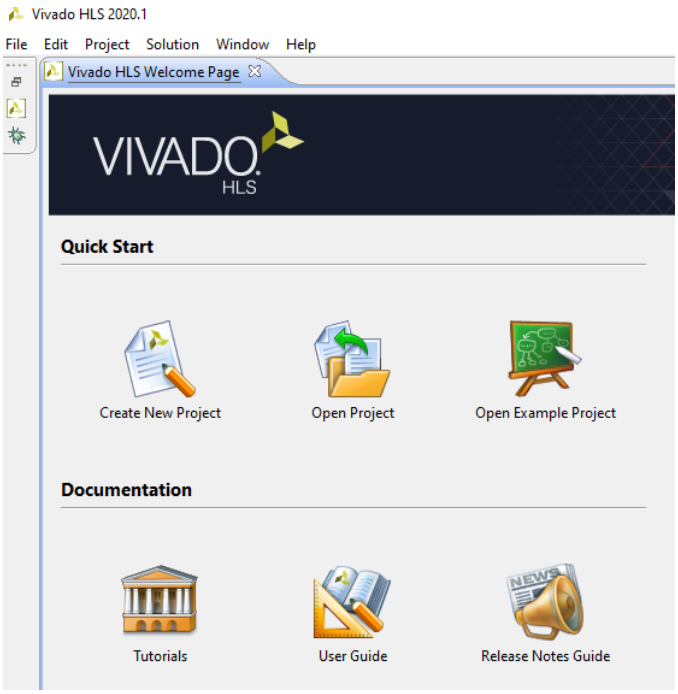


Figure 12. Vivado HLS homepage

2. Select the project name and the location.
3. Enter “add” as top function then click “New File”.
4. Select the file name with extension “.cpp” and save. (Example file name: adder.cpp)
5. The file you just created should be listed then click “Next” .
6. Click “Next” again. (Note: There are no TestBench files.)
7. Click the “…” button on the right to select part.
8. Enter “xc7z020clg400-1” in the search box which corresponds with the PYNQ-Z1, select it from the list, and click “Ok”.
9. The part you just selected should be in bold then click “Finish”.
10. Now the project has been created. Under the Explorer tab that is open, expand “Source”, and double click the .cpp file to open the source code in the right window. Refer to Figure 13.

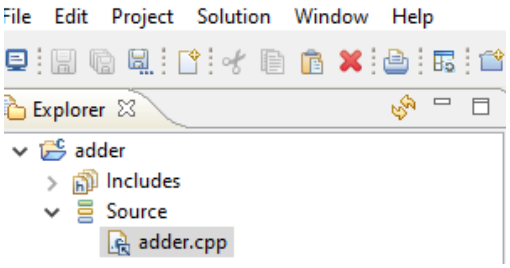


Figure 13. Source is expanded to view the file.

11. Copy and paste the following code from the overlay tutorial page and refer to Figure 14 [13]:

```
void add(int a, int b, int& c) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=c

    c = a + b;
}
```

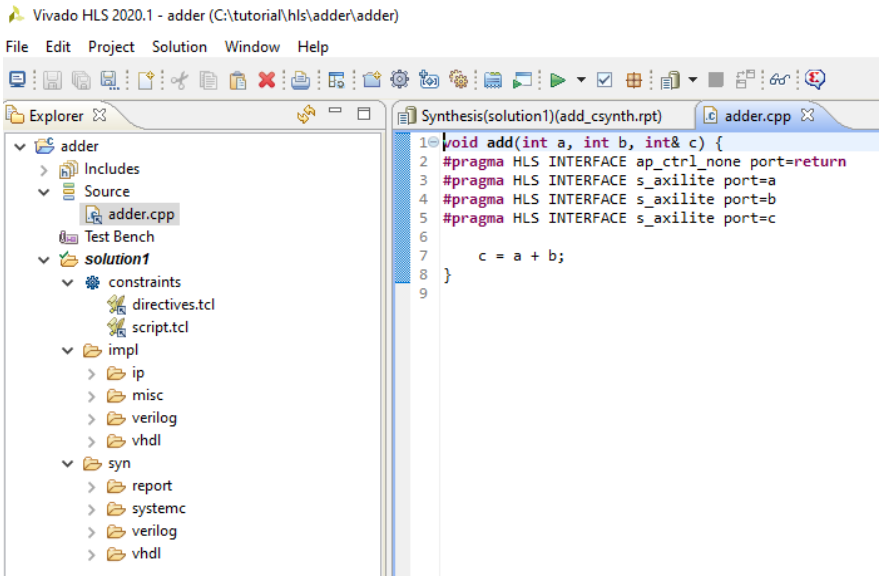


Figure 14. The full source code added to the .cpp file.

12. Click the “floppy disk” button to save the code to the file.

13. Click the "play" button above the editing window to run HLS' C synthesis, which will convert the code to HDL. Refer to Figure 15.

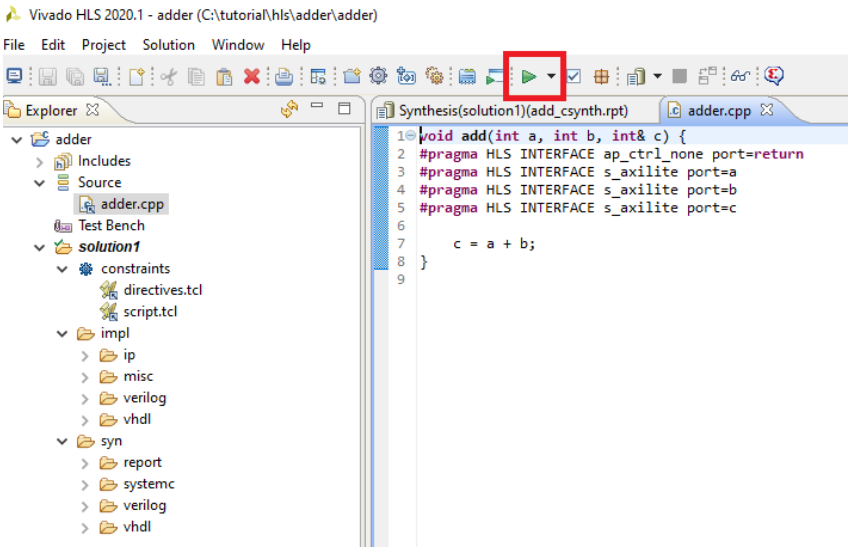


Figure 15. The button highlighted with the red box will run the synthesis and convert the code to HDL.

14. Click the “orange window box” button to export RTL then click “Ok” on the pop-up box. Refer to Figure 16. (Note: Look under either the Verilog or VHDL files in '/impl' to find the memory addresses used in the write/read functions of the overlay. These addresses can also be found in the other Vivado software during IP integration.)

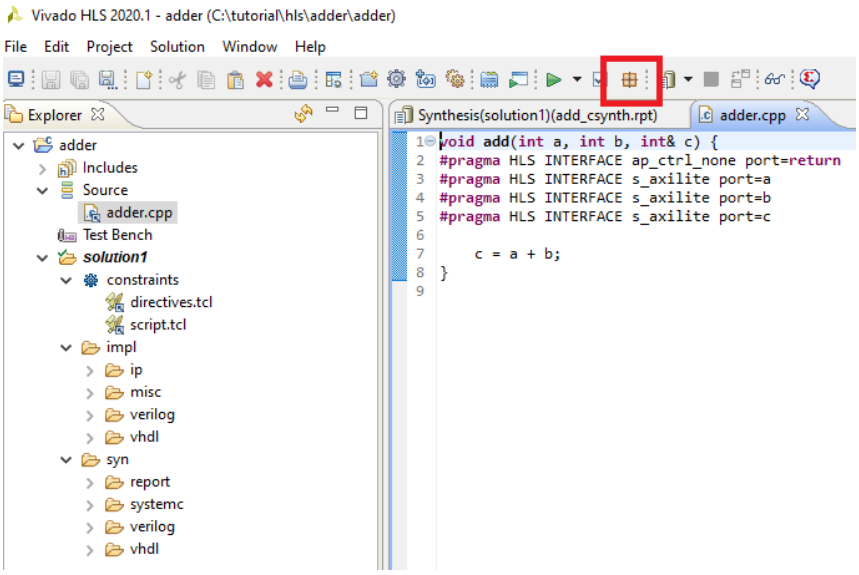


Figure 16. The button highlighted with the red box will export RTL.

15. Open Vivado then click “Create Project” on the homepage under Quick Start then “Next”. Refer to Figure 17 and Figure 18. (Note: Everything needed for the block to be imported in Vivado has been done)

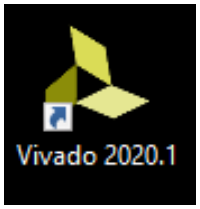


Figure 17. Vivado icon

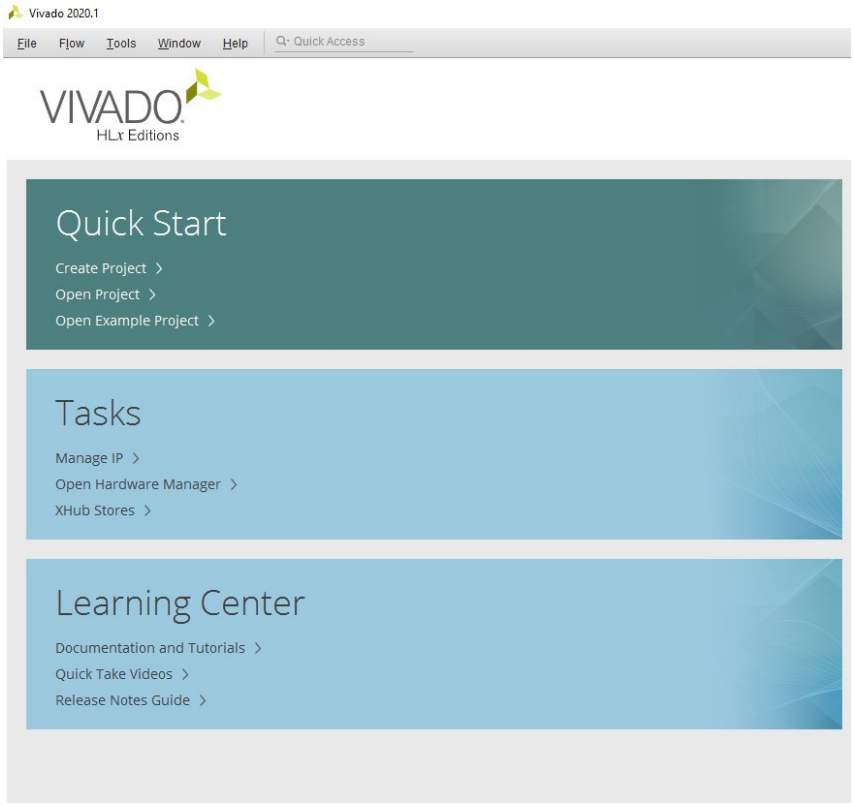


Figure 18. Vivado homepage

16. Select a project name and location then click “Next”. Refer to Figure 19.

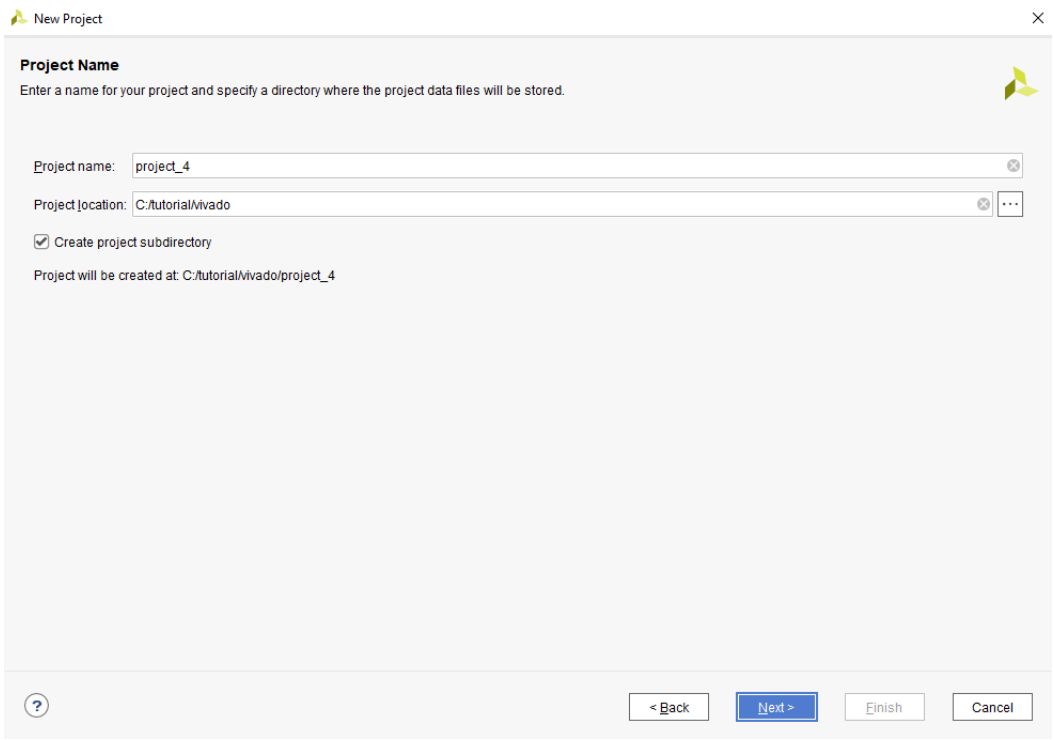


Figure 19. Page to select project name and location.

17. Select “RTL project” and click “Next”. Refer to Figure 20.

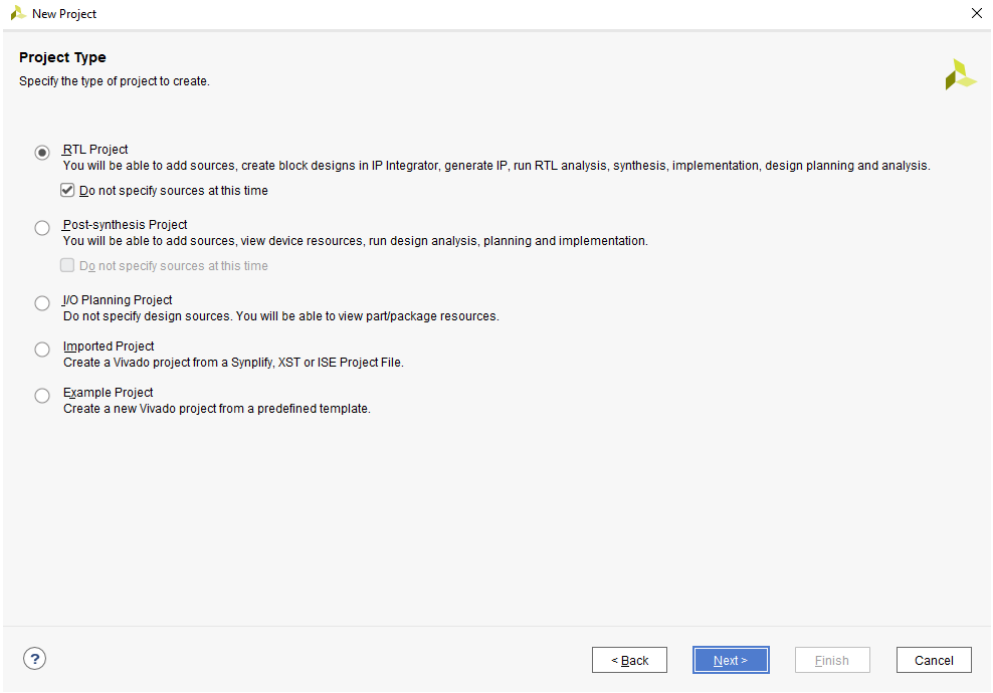


Figure 20. Page to select project type.

18. Click the Boards tab, select “PYNQ-Z1”, and click “Next”. Click “Finish” on the last page. Refer to Figure 21.

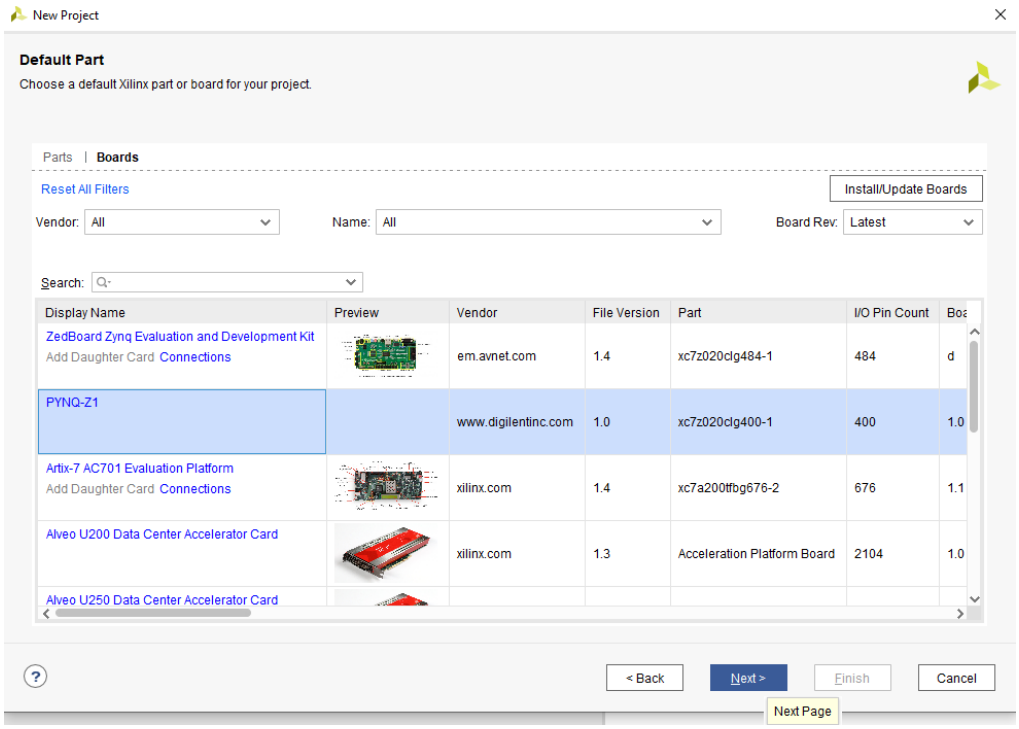


Figure 21. The PYNQ-Z1is highlighted under the Boards tab.

19. On the left under “IP INTEGRATOR”, click “Create Block Design”. Refer to Figure 22.

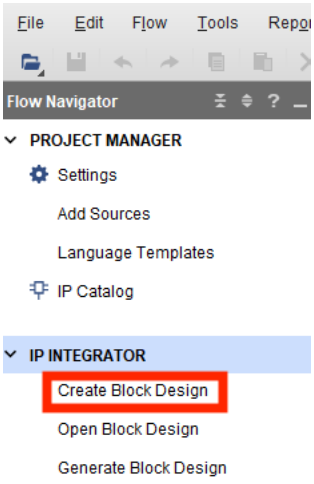


Figure 22. The phrase highlighted with the red box will create the block design.

20. Choose a name or leave the default name and click “Ok”.

21. Click the “+” button and scroll to the bottom to select ZYNQ7 Processing System. Refer to Figure 23.


Figure 23. The button highlighted with the red box will add to the block diagram.

22. Click “Run Block Automation” then “Ok” on the pop-up box.

23. On the toolbar, click “Tools” then “Settings”. Expand IP under project settings and click “Repository”.

24. Click the “+” button, find the HLS project that was previously created, click “Select”, and lastly, click “Ok” twice.

25. Click the “+” button under the Diagram tab. Type “add” in the search bar and select “Add”.

26. In the Block Properties box on the left, change the name to “scalar_add”.

27. Click “Run Connection Automation” then click the “floppy disk” button to save the design.

28. In the window to the left of the block design, click the Sources tab.

29. Right click on the name of the design under Design Sources, choose “Create HDL Wrapper” then click “Ok”. Refer to Figure 24.

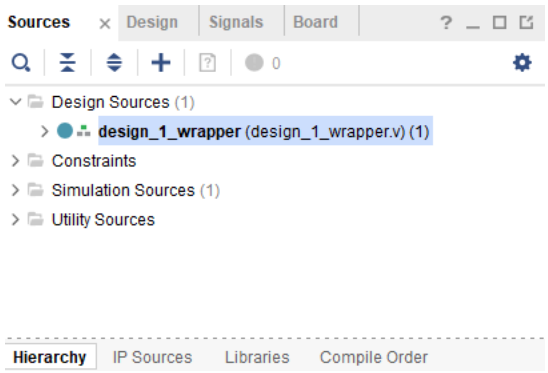


Figure 24. The wrapper has been created.

30. In the far-left window, expand Program and Debug at the bottom to click "Generate Bitstream" then "Yes". Refer to Figure 25.

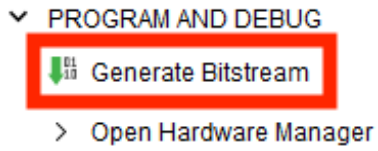


Figure 25. The phrase highlighted with the red box will generate the bitstream.

31. Under File in the toolbar, choose "Export", click "Export Block Design" then click "Ok" on the pop-up box. Refer to Figure 26.

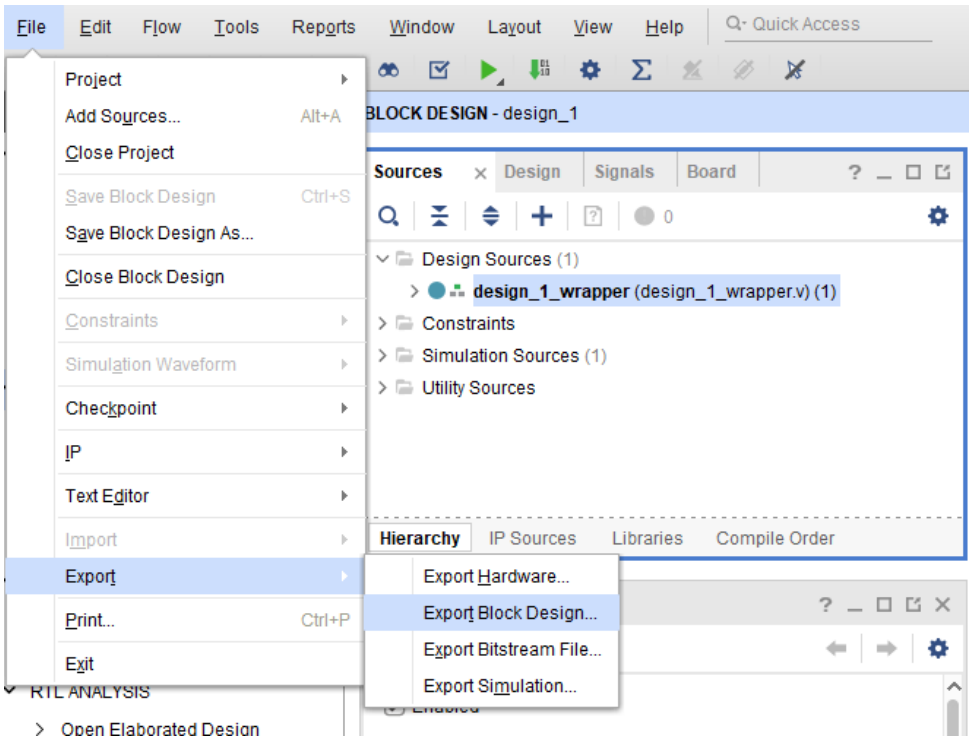


Figure 26. Export the block diagram.

32. Rename the tcl and bit files to match the cpp file name

33. Copy both tcl and bit files to your PYNQ board under the overlays folder.

34. On the right side of the board homepage, select “New” then “Python 3” to open a new Jupyter notebook. Refer to Figure 27.

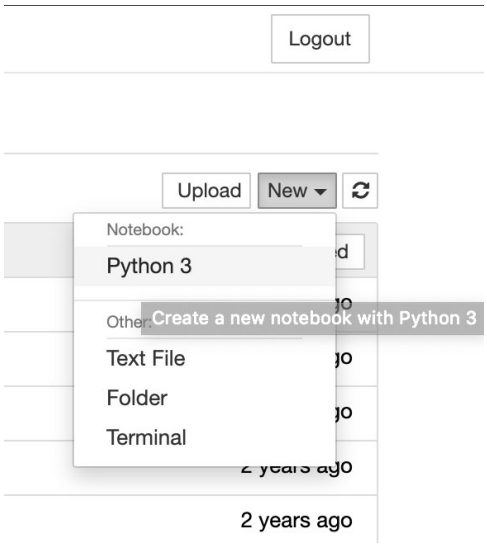


Figure 27. Open a new Jupyter notebook.

35. In block 1, paste the following code from the tutorial page [13]:

```
from pynq import Overlay
overlay = Overlay('/home/xilinx/tutorial_1.bit')
```

36. Replace the '/home/xilinx/tutorial_1.bit' string in the above code in step 35 with a string of the path to your .bit file on the PYNQ board.

37. Click the “Run” button.

38. In block 2, paste the following code from the tutorial page then click the “Run” button [13]:

```
overlay?
```

39. In block 3, paste the following code from the tutorial page then click the “Run” button [13]:

```
add_ip = overlay.scalar_add
add_ip?
```

40. In block 4, paste the following code from the tutorial page then click the “Run” button [13]:

```
add_ip.write(0x10, 4)
add_ip.write(0x18, 5)
```

add_ip.read(0x20)

41. The answer of 9 will output from the addition of the registers. Figure 28 shows the blocks of code and the output mentioned in steps 35 and 38-41.

```
In [1]: from pynq import Overlay

        overlay = Overlay('/home/xilinx/pynq/overlays/adder/adder.bit')

In [2]: overlay?

In [3]: add_ip = overlay.scalar_add
        add_ip?

In [4]: add_ip.write(0x10, 4)
        add_ip.write(0x18, 5)
        add_ip.read(0x20)

Out[4]: 9
```

Figure 28. The full code with output.

42. Optional: For additional experimentation, change the inputs for the registers in block 4 then click the “Run” button for alternative outputs.

## F. LAB SUMMARY

In this lab, a custom IP was created in Vivado HLS. That IP was converted to a bitstream targeting the PYNQ-Z1 board in Vivado. That bit stream was moved to the board, and subsequently tested successfully in the PYNQ-Z1's built-in Jupyter notebook programming environment.

## G. LAB QUESTIONS

1. Does changing the Python code change the behavior of the circuit?

- Does changing the operator from addition to subtraction change the computed result?

- Does changing the values of one or both operands change the computed result?

2. Did you successfully reach the end of the lab?

- Did you encounter any errors or issues at any stage of the lab?

3. Did you conduct any extra experimentation or exploration? If so, please describe.

4. What did you learn from this lab?

5. How could this lab be improved?

6. Do you have any ideas of other types of computations that this system can accelerate efficiently?

## H. CHAPTER SYNOPSIS

This chapter examined the implementation of the Xilinx ecosystem. It covered the configuration of the PYNQ-Z1 FPGA board including the install of the board files and the install of the Vivado design tools. It also included a detailed explanation of the bitstream creation and execution.

The next chapter discusses the conclusion and future work.

THIS PAGE INTENTIONALLY LEFT BLANK

# V. CONCLUSION AND FUTURE WORK

This chapter summarizes the research performed and proposes additional avenues to further the research.

## A. CONCLUSION

This research answered the following question:

### 1. Research Question

*How does the technology of the Xilinx ecosystem, composed of the PYNQ board along with Vivado design software suite, allow a computer science student to leverage parallelism to accelerate a program, in lieu of any electrical engineering training*?

This thesis contributes a way for computer science students to learn how to leverage parallelism in order to accelerate their applications due to the drastic slowing of sequential CPU performance. Custom hardware like FPGAs is capable of accelerating high-throughput applications by leveraging the parallelism of billions of transistors provided by Moore's Law. FPGAs are an important part of this parallel computing landscape that ensures application performance continues to improve while offering economic tradeoffs in comparison to other custom hardware options.

The goal of this research was to combat traditional FPGA development, which requires years of electrical engineering training with extensive knowledge of HDL or VHDL and difficult lab environments with rigid tools that only work on Windows platforms by utilizing the Xilinx University ecosystem.

This thesis demonstrated the technical implementation of the Xilinx ecosystem, which consists of a low-cost FPGA board, the free Vivado suite of design tools, and an intuitive programming language, via a simple lab. The Xilinx ecosystem and architecture allows a student to write a simple Python program that adds two numbers, and this program is converted by the tools into a working circuit that runs on the Xilinx FPGA device. This thesis presents a pathway for computer science students to develop familiarity with and

begin working in the Xilinx ecosystem, in which they will be able to garner hands-on experience with real hardware.

## B. FUTURE WORK

This research focused on the initial steps for implementing the Xilinx ecosystem via a simple example. There are a host of possibilities for future work. The following recommendations focuses on expanding the usability of the Xilinx ecosystem:

### 1. Creating a driver

This research used the default "UnknownIP" driver, which according to Xilinx is "useful for determining that the IP is working it is not the most user-friendly API to expose to the eventual end-users of the overlay." [13] Xilinx provides a tutorial to create a driver to accompany the example utilized in this research. Future work could encompass creating a driver specific to this research example.

### 2. Overlay customization

Custom overlays are used to enhance the user experience and are used in conjunction with the host of drivers included in the PYNQ library. Future work would include exploration of overlay creation and PYNQ integration of that new overlay.

# LIST OF REFERENCES


[1] K. Asanovic et al., "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[2] J. Cavanagh, *Verilog HDL*. Boca Raton, FL, USA: CRC Press, 2007.

[3] D. Thomasset and M. Grobe, *Introduction to the Message Passing Interface (MPI) using C*, The University of Kansas, November 2013. [Online] Available: http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml

[4] A. Wong, *Create an FPGA-based IoT application by Python*, June 1, 2020. [Online]. Available: https://www.rs-online.com/designspark/iot-remote-monitoring-using-pynq-framework

[5] S. Hauck and A. DeHon, *Reconfigurable Computing*. Boston, USA: Morgan Kaufmann, 2008.

[6] J. Hennessy and D. Patterson, *Computer Architecture*. Cambridge, MA, USA: Elsevier, 2019.

[7] H. Huff and D. Gilmer, *High Dielectric Constant Materials*. Berlin: Springer, 2005.

[8] R. Kastner, J. Matai, and S. Neuendorffer, *Parallel Programming for FPGAs.* 2018.

[9] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *FPGA 2006*, Monterey, CA, February 2006. [Online] Available: https://www.eecg.utoronto.ca/~jayar/pubs/kuon/kuonfpga06.pdf

[10] M. McCool, A. Robison, and J. Reinders, "Introduction," in *Structured Parallel Programming*. San Francisco, CA, USA: Morgan Kaufmann, 2012, pp. 1–38.

[11] M. Oskin, "The revolution inside the box," *Communications of the ACM*, vol. 51, no. 7, pp. 70–78, 2008.

[12] A. Putnam et al., "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, 2015.

[13] Xilinx Inc., PYNQ Introduction — Python Productivity for Zynq (Pynq). 2020. [Online] Available: https://pynq.readthedocs.io/en/v2.5.1/.

[14] J. Rose et al., "Architecture of FPGAs: The Effect of Logic Block Functionality on Area Efficiency," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 5, October 1990.

[15] P. Ross, “5 Commandments [technology laws and rules of thumb],” *IEEE Spectrum*, vol. 40, no. 12, pp. 30–35, 2003.

[16] J. Villasenor and W. Mangione-Smith, “Configurable Computing,” *Scientific American*, vol. 276, no. 6, pp. 66–71, 1997.

[17] Xilinx University. 2020.[Online] Available: https://www.xilinx.com/support/university.html

[18] FPGA Developer, “How to make a custom PYNQ overlay,” *YouTube,* March 15, 2018. [Online] Available: https://www.youtube.com/watch?v=Dupyek4NUoI

# INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California