EXPLORATION OF FPGA BASED ACCELERATORS IN LINUX HOST SYSTEMS

by

Daniel May

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Spring 2021

EXPLORATION OF FPGA BASED ACCELERATORS

IN LINUX HOST SYSTEMS

by

Daniel May

Approved: _____

Fouad Kiamilev, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Jamie D. Phillips, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____

Levi T. Thompson, Ph.D.
Dean of the College of Engineering

Approved: _____

Louis F. Rossi, Ph.D.
Vice Provost for Graduate and Professional Education and
Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to thank Dr. Chase Cotton for his immense support and advisement throughout this project. Additionally I would like to thank my family for their support of my academic endeavors.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

PCIe devices are pervasive in today's computer systems. In addition, FPGAs are increasingly finding use in data centers as accelerators for specialized computation tasks. This paper endeavors to provide a guide for future researchers to use as a springboard for future work in this area by providing working examples and documentation for various FPGA designs and host system drivers.

Chapter 1

INTRODUCTION AND MOTIVATION

Early computers such as the Atanasoff–Berry and Colossus computers were highly specialized to perform a particular task. Over time the field of computing moved toward general-purpose computers built around the von Neumann architecture. However, there has recently been revived interest in non-von Neumann architectures for specialized computing tasks with the added challenge of seamlessly integrating such special-purpose computers with existing von Neumann systems and infrastructure.

1.1 Introduction

Modern computing relies heavily on peripheral devices such as GPUs and FPGAs for applications such as machine learning and rendering applications. In order for these devices to work in an effective manner, they require a method of efficiently moving data between the host and the device and between multiple devices. The most common protocol used for this task on modern systems is PCIe, Peripheral Component Interconnect express.

FPGAs, Field Programmable Gate Arrays, are common in many applications that require massively parallel computation and/or have real-time constraints. FPGA use cases have overlap with ASICs and are especially useful in applications with low production volumes where custom digital logic is required, but it would be financially impractical to fabricate a dedicated ASICs. Additionally, FPGAs can be used in coprocessor designs for machine learning a digital signal processing. Many such applications also require a high bandwidth connection to a server or workstation resulting in the rise of FPGA-based PCIe accelerator cards.

In order to take advantage of these accelerators, substantial development work must be done both to design the FPGA firmware and the operating system driver for interfacing with the card. This thesis will focus on FPGA-based PCIe expansion cards and how to interface them with a Linux host computer with an emphasis on the practical details of implementing such a system.

1.2 Motivation

FPGA accelerator cards have the potential to make a significant impact on many fields and industries. However, the lack of straightforward documentation regarding how to achieve this goal is holding back the widespread adoption of FPGA accelerators. This is at least partly due to the fact that both in-depth knowledge of both the Linux kernel and FPGA designs are required in order to properly implement a successful design. As a result, there is often no clear path by which an FPGA accelerator can be used in a specific application, thus increasing the risk of making significant investments of time and resources with no guarantee of success.

The goal of this thesis is to present an implementation of both a driver and FPGA design that can be used as a reference for future development work. As such the following priorities have been established:

1. The design must be easy to modify. Since the use case of the end-user is unknown, the design should be generic enough so it can be easily modified and extended.
2. Use low-cost parts. In order to ensure accessibility for the widest range of potential users expensive components are to be avoided when possible.
3. Well documented. Since the objective is for the end-user to be able to extend the design to meet their needs, it must be easy for the user to understand all aspects of the reference design.

Chapter 2

TECHNOLOGY SUMMARY

As with all modern engineering endeavors, the results presented here would not be achievable if it were not for the extensive collections of tools and knowledge provided by those who came before. As such, it is fitting that prior to exploring the primary topic of this thesis a moment should be taken to discuss the existing work and research that forms its foundation.

2.1 Linux Overview

Linux is a monolithic operating system kernel developed by Linus Torvalds starting in 1991 [14, p. 85]. It is now widely used in applications ranging from embedded systems to supercomputers, with billions of Linux systems currently in use around the world [7, p. 158]. Linux itself is nothing more than an OS Kernel and relies heavily on software and libraries provided by the GNU project in order to function as a complete and usable operating system [13]. Examples of such software are the GNU Compiler Collection (gcc), the GNU C Library (glibc), and the Bourne Again Shell (bash) [11]. The Linux Kernel is licensed under the GNU Public License Version 2 [6], which allows for users to modify and redistribute the code, with the requirement that distributed changes be licensed in the same manner [10]. There is debate regarding the naming of Linux, with some simply calling it Linux and others calling it GNU/Linux [13]. For the sake of brevity, this document will simply refer to the operating system as Linux.

In order to fully understand Linux, it is important to first discuss Unix. Unix is an operating system initially developed at Bell Laboratories in the late 1960s and 1970s and quickly spread throughout academia and industry [7, pp. 34, 87, 131, 144].

Beyond just being an operating system, Unix provided a design philosophy which Doug McIlroy, a researcher at Bell Labs, summarized as follows [12, pp. 11, 12].

1. "Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features."
2. "Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input."
3. "Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them."
4. " "Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them."

Linux is a Unix-like operating system, often referred to as *nix-like due to copyright reasons. The term "Unix-like" refers to operating systems that share the same structure and design philosophy as the official Unix operating system while not being directly related to official Unix. Other prominent Unix-like operating systems include The Berkeley Software Distribution, often abbreviated BSD, and Minux [7, pp. 153, 158].

2.2 Linux Kernel Driver Model

The Linux Kernel allows support to be added for hardware components through device drivers. These drivers run as part of the kernel and as such receive privileged access to system resources, such as attached devices. Since drivers run as part of the kernel itself, they have the advantage of being allowed access to all of the kernel's internal APIs. The downside is that drivers are not allowed to make use of any user space libraries. This alone makes kernel development very different from user space programming as commonly used functions, and structures will not be available. Additionally, driver developers have the added responsibility of being sure that their drivers will be well behaved since, unlike in user space where a poorly behaved program will

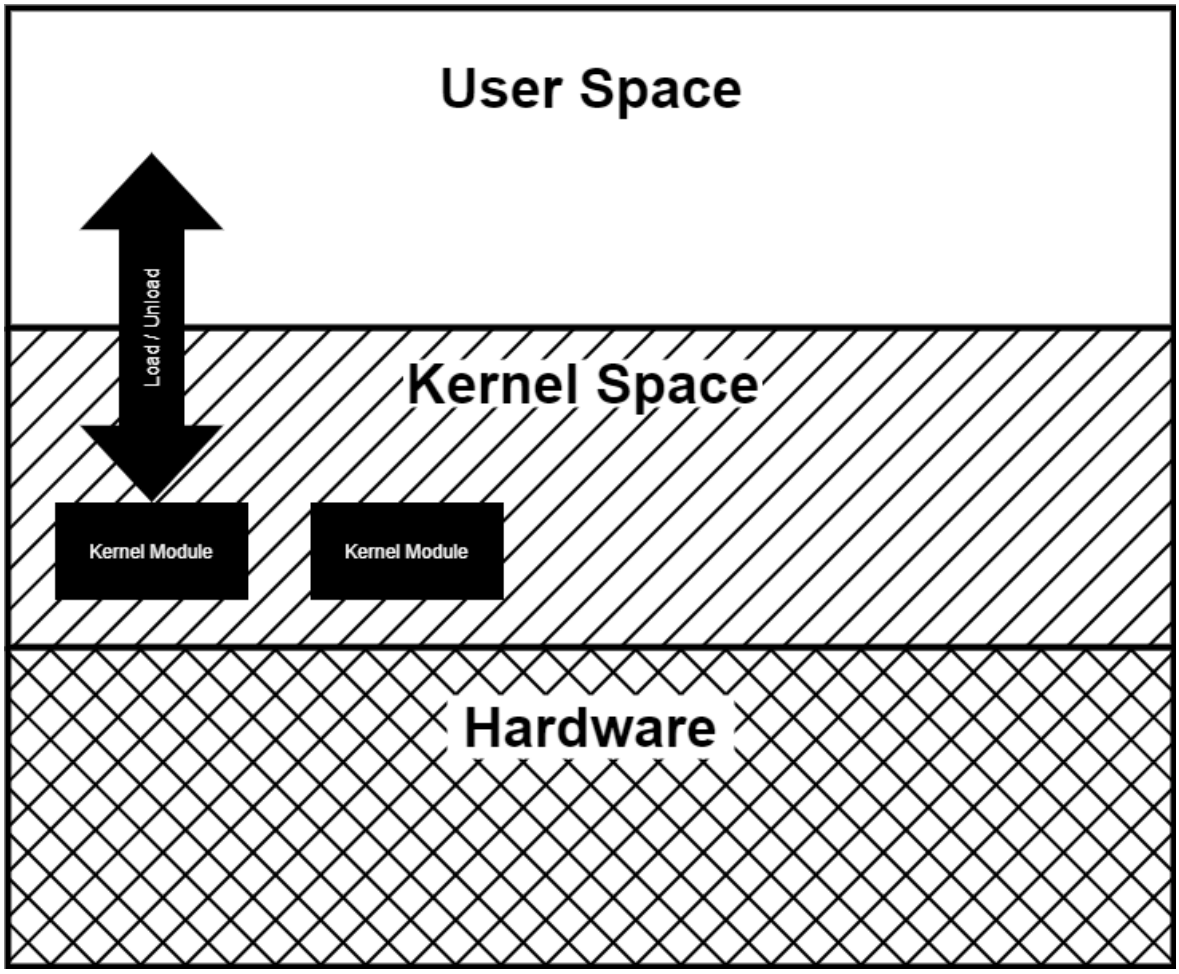

Figure 2.1: Linux System Overview

itself crash but leave the rest of the system untouched, a badly behaving driver has the potential to bring down the entire system.

While some changes to the Linux Kernel require that the kernel be recompiled, a time-consuming process, it is also possible to compile drivers separately and load and unload them from a running kernel, as demonstrated in figure 2.1. this is the most common method used for driver development.

2.3 PCIe Overview

In order to understand PCIe it is helpful to first discuss the PCI protocol. PCI stands for Peripheral Component Interconnect [4] and provides a parallel shared bus for communication between hardware components [8]. PCI was later replaced by PCIe, Peripheral Component Interconnect express, which handles communication in a serial rather than parallel manner [4]. Another key difference between these protocols is how the handle interrupts. PCI has a fixed number of interrupt lines that are shared between all of the devices on the bus [3] while PCIe on the other hand uses Message Signaled Interrupts (MSI) which communicate through special addresses used to trigger interrupts on the host system [5]. This means that there is no longer a need to share interrupt lines between various cards as was previously required with PCI. The primary method of passing data back and forth between the host and the device are Base Address Regions. These are essentially memory regions on the device that can be accessed by the host system. PCIe provides a high-bandwidth connection with various peripheral cards that can be used to extend the capabilities of the host system.

2.4 AXI Interface

The Advanced eXtensible Interface (AXI) is a bus protocol with multiple variants and versions [2]. It is a part of the Advanced Microcontroller Bus Architecture (AMBA) developed by ARM for connecting logic blocks within SOCs [2]. It has since been used by Xilinx as a method of connecting IPs within FPGA designs [15]. The variants of the protocol are AXI4, AXI4-Lite, and AXI4-Stream [2]. The AXI4 interface is a transaction-based interface [1, p. 24] with AXI4-Lite being a subset of AXI4 [1, p. 22]. AXI has four communication channels, as specified in [1, p. 25], are listed below:

- Read Data Channel
  - Carries the data being read from the IP.
- Read Address Channel
  - Carries the address to be read from on the next read command.

- Write Data Channel
  - Carries the data being written to the IP.
- Write Address Channel
  - Carries the address to be written to on the next write command.
- Write Response Channel
  - Carries the response from the IP following the completion of a write.

One of the key differences between AXI4 and AXI4-Lite regards burst transactions. AXI4 allows data to be transmitted and received in bursts of up to 256 individual data transfers [1, p. 46]. These bursts increase efficiency by not requiring an address to be provided for every data transfer and will instead compute the address based on the selected burst mode. The first of these burst modes is **Fixed** mode, where every data transfer goes to the same address [1, p. 47]. In **Incremental** burst mode, each subsequent address is equal to the previous address incremented by the data transfer width [1, p. 48]. Last is **Wrap** mode, which restarts the address at the beginning of the address range when the end of the range has been reached [1, p. 48]. Besides this difference, Wrap mode generally operates in a similar manner to Incremental mode [1, p. 48]. By contrast, AXI4-Lite can only handle bursts of a single data transfer [1, p. 126] which decreases its efficiency but makes it simpler to implement.

In addition to AXI4 and AXI4-Lite protocols, there is AXI4-Stream. AXI4-Stream mode does not have any context of addresses to go along with the data [18, p. 9]. This allows for higher efficiency connections and is useful in applications where data is directly streamed from one device or IP to another [18, p. 9].

All AXI protocols use a handshake when transferring data that is synchronous to the bus clock [1, p. 39]. Each channel has a **ready** signal that is controlled by the receiver, and a **valid** signal, controlled by the transmitter [1, p. 39]. When the transmitter has data to send, it will set the **valid** signal high [1, p. 39]. Likewise, when the receiver is able to receive data, it will set the **ready** signal high [1, p. 39]. When

both **ready** and **valid** are high, the handshake has been completed, and the data can then be transferred over the channel [1, p. 39].

2.5 PCIe and Linux

The Linux Kernel provides various APIs to make interfacing with groups of devices easier. Of particular interest is the PCI API, which is used to set up and communicate with PCI and PCIe devices. Despite being named after PCI, this API is also used when working with PCIe devices. The primary goal of this API is to provide a way to communicate with PCI and PCIe devices in a manner that is independent of how the underlying architecture implements this functionality [9].

2.6 Xilinx PCIe IP

Many Xilinx FPGAs contain a hardware block for specifically for handling PCIe communication [17]. In order to make use of this block, that is built into the board's Axtix 7 FPGA, the AXI Memory Mapped to PCI Express IP is used. This IP allows the FPGA design to communicate with a host system over the PCIe interface by converting the incoming PCIe commands received by the hardware block into AXI4 packets that can be used by various IPs within the FPGA logic [16, p. 6]. The core provides various customization options making it easy to tweak for use in specific situations. On the PCIe side, there is an option to have up to 3 BARs [16, p. 8], each of which has an adjustable memory size [16, p. 72]. BAR stands for Base Address Register and is a memory region on a PCIe device that is accessible by the host system. Additionally, it is possible to set static values such as Vendor and product IDs that can be queried by the host system [16, p. 70]. Controls are also provided to trigger interrupts on the host system from within the design logic. These interrupts can either be of the MSI variety or emulated legacy interrupts [16, p.56]. On the AXI side there are options to control the width of both the addresses used and the width of the AXI read and write data buses [16, p. 76].

2.7 Existing Work

Interfacing with PCIe devices is a common task in industry and, as a result, is well understood within the organizations that leverage this technology. However, gaining the necessary prerequisite knowledge to begin working in this field can be challenging. Likewise, the subject of driver development is less commonly covered than many other programming topics due in part to its specialized nature. The most complete resource on the topic is the book *Linux Device Drivers*, Third Edition by Corbet, Rubini, and Kroah-Hartman, which covers the topic of driver development in great detail and is an invaluable source of information on the subject. However, this book has a few shortcomings. First, it is extremely information-dense, which can make it difficult to approach as a beginner. Second, since it was published in 2005 some of its information is out of date due to kernel ABIs changing and new technologies becoming prominent such as PCIe.

The other aspect of this research area is firmware development for the FPGA card itself. Documentation and PCIe-based example designs are often provided by the FPGA vendor, such as Xilinx, in the case of the hardware selected for this project. These resources usually consist of example designs and corresponding drivers along with datasheets for the PCIe IP module. These materials are key since they provide a working design to compare against and build off of. However, they do a poor job of explaining why certain design or configuration choices were made and can leave beginners with no clear path forward to understand how to modify the example to meet their needs or build their own design from scratch.

Chapter 3

DEVELOPMENT WORK AND RESULTS

In this chapter the details of the reference designs will be discussed in depth. The goal is that this chapter, combined with the reference design source code, will provide future users with the knowledge needed to develop their own designs and drivers.

3.1 Hardware Used

The central part of this project is the FPGA accelerator card that must be capable of connecting to a host PC over PCIe. There are many such cards available from multiple vendors with a wide range of prices and capabilities. From the array of options, the LiteFury board from RHS Research LLC, based on the Xilinx Atrix 7 FPGA, was chosen. The LiteFury board costs less than $200, making it an affordable option accessible to most researchers and students. The LiteFury can be installed in either an m.2 slot or a standard PCIe slot, though the latter requires an additional adapter board, which makes it compatible with most desktop PCs and workstations. Peripherals on the board are somewhat limited due to its focus on data acceleration, however, it still manages to include 256 megabytes of on board RAM and provides 12 general purpose I/O pins, which can be configured to provide two analog input channels, along four separately addressable LEDs. The board can be programmed through a PicoEZmate connector using standard Xilinx programming tools through the included PicoEZmate to a 14-pin Molex passive converter board.

3.2 LED Control Project

The goal was to create a very basic project to act as a Hello World program for accelerator card development. The design is divided into a front-end that communicates

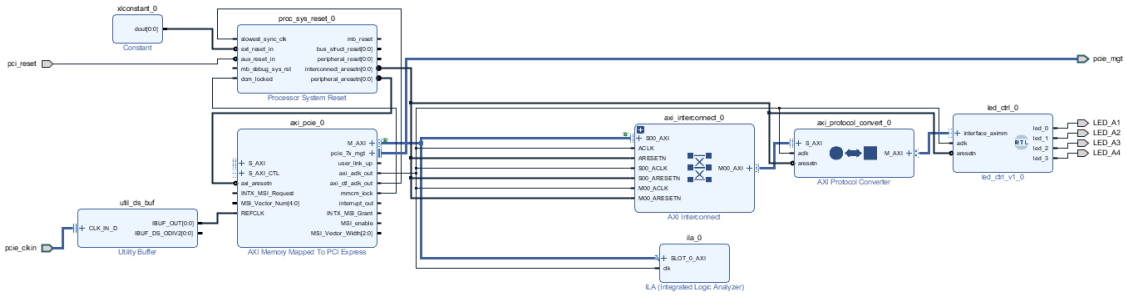

Figure 3.1: Simplified block design for the LED Control project

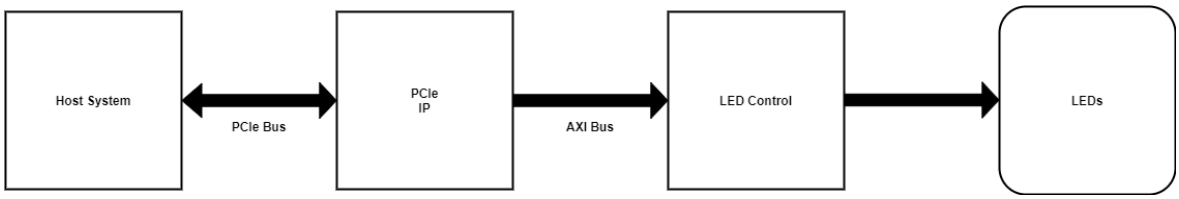

Figure 3.2: Block design from the Vivado IP Integrator for the LED Control project

using an AXI bus, and a back-end that is responsible for directly controlling the LEDs.

3.2.1 LED Control Front-End

The front-end design was constructed primarily using the Block Design Editor in the Xilinx Vivado IDE, as shown in figure 3.1. The Block Design Editor allows for easy integration with existing Xilinx IP in an intuitive graphical environment. A simplified version of this block design is presented in figure 3.2. The AXI Memory Mapped to PCIe IP, referred to from here on as the PCIe IP for simplicity, outputs read and write requests on an AXI4 bus which correspond to reads and writes with in the PCIe BAR0 as discussed in section 2.6. Assuming that no address translation is used in the PCIe IP, a write to BAR0 address 0x15 will be converted to an AXI write command to address 0x15. If address translation was used, then the translation value

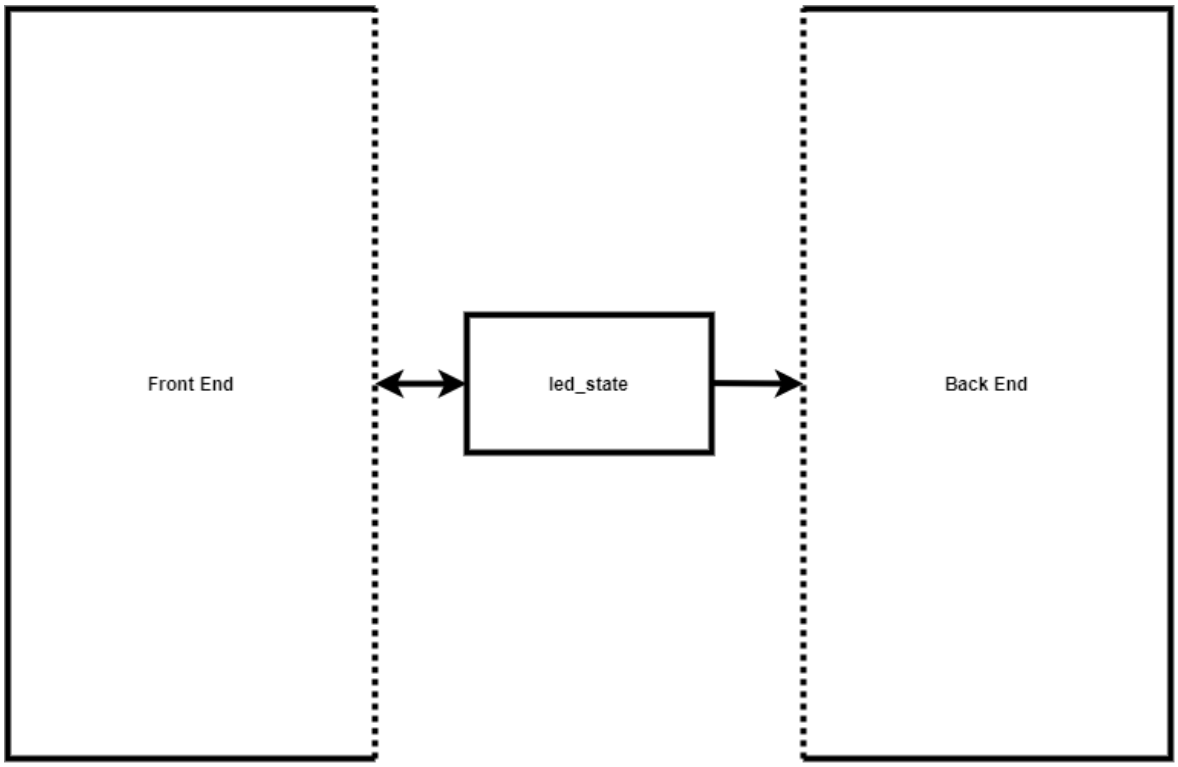

Figure 3.3: LED-Control Front-end and back-end communication registers.

will be added on to the BAR address. For example if an address translation value of 0x50 was used then a write to BAR address 0x15 would be sent to AXI address 0x65.

For simplicity the LED Ctrl block implements an AXI4 Lite interface and therefore requires a protocol converter to be connected in between it and then PCIe IP since the PCIe IP provides an AXI4 interface. What is not clearly shown from the diagram is the AXI4 Lite slave RTL block internals. This block has a designated AXI address range that it will respond to that is set in the code. These addresses correspond to internal registers that are used to communicate with the prime finder back-end logic. The relationship between the front-end and back-end of the design is shown in figure 3.3.

3.2.2 LED Control Back-End

The back-end of the LED control design is very simple. The first four bits of the value stored in the register are each broken out as individual signals and routed out of the module. Now that these signals are available in the block design, they can be connected to top-level ports for the LEDs. Due to this simplicity, it is not split into its own module in the design and is instead placed directly alone side the front end.

3.2.3 LED Control Driver

The driver allows the Linux host to communicate with the firmware running on the FPGA and is implemented as a loadable kernel module. In order to make development easier, additional scripts are provided to allow the FPGA to be reprogrammed without needing a full system reboot. The driver itself can be broken down into the following subsections.

- PCIe mapping and control
- File operations

Each of these will be covered in detail below.

3.2.4 PCIe Mapping

In order to communicate with the FPGA, the driver must establish a connection with the board over PCIe. The task of actually pairing the driver with the device is handled by the kernel and not the driver itself. To facilitate pairing, the driver must provide information to the kernel regarding which devices it can work with. This is done by providing the kernel with an array of pci_device_id structures, each of which contains the vendor and device IDs of compatible devices. The data is then provided to the kernel using the MODULE_DEVICE_TABLE macro.

Now that the kernel knows what devices to pair with the driver, functions must be provided for the kernel to invoke when a device is paired with the driver. The minimum functions are the **probe** and **remove** functions. The **probe** function is called when a device is paired with the driver and allows for configuration and setup to occur. In this case, there are several important tasks that must be handled. First, the device must be enabled by calling the pci_enable_device function call. Once the device is enabled, it is possible to read from it such as values from its configuration space and the start and end addresses of its Base Address Registers (BARs). When the start and end addresses of a BAR have been acquired, the size of the BAR can be found, and it can be mapped into a virtual address space using. This virtual address space mapping allows the BAR of the device, which has a physical address that is not directly accessible, to be mapped in the process's virtual address space, thus allowing that process to access this memory region like any other memory buffer. The kernel function to handle this mapping is ioremap which takes in a physical starting address and the size of the region to map, both of which acquired from the device previously, and will return a pointer that is usable by the calling process.

The PCI **remove** function is called when the driver exits and should generally undo the configuration done in the **probe** function. For the prime finder driver this means un-mapping the device's physical addresses, using iounmap, and disabling the device using the pci_disable_device function.

3.2.5 File Operations

Files in the /dev directory provide a way for user space programs to communicate with device drivers. In order to support such communication, a driver must implement and register a set of file operation functions that will be invoked by the operating system when the corresponding system call is invoked in user space. The most basic of these functions are open and close. As would be expected, the open function is called when the device's file is opened, and the close function is called when the device's file is closed. These functions provide a good place to setup and teardown anything that is only required when the driver is being interacted with. In general, this does not include device configuration since this can persist between opening and closing the device file.

The driver internally tracks the current position within the file as an offset from the start. This value can be set or modified from user space using the llseek function. There are two ways to set the offset. The first is in absolute terms where the specified offset is relative to the start of the file. The second alternative is to move the offset relative to its current position. In this driver, the offset value is used to control where data is read or written to in the device's BAR0.

Unsurprisingly, there are also read and write functions that actually facilitate data transfer between user space and the driver. In each case, a buffer, and its size are provided so data can either be read from it or written to it depending on the direction of the transfer that is occurring. Special care must be taken when working with these buffers since the buffer is located in user space memory and cannot be safely accessed directly from kernel space. To get around this limitation, a buffer of matching size should be allocated in the driver. Data can then be copied to/from user space into this kernel buffer using either the copy_from_user or copy_to_user functions, which allows for safe data transfers to occur. An additional function parameter not previously mentioned is the offset within the device file. The topic of offsets will be discussed below when the read and write implementations are covered below.

The prime finder driver uses the read and write functions to allow user space

programs to transfer data to/from the LiteFury boards BAR0. In the case of the write function, a pointer to a buffer in user space is provided that contains the data to be written. This data is copied to kernel space before being written to the LiteFury's memory using the iowrite32 function. Of particular note is the address that the data is written to. As stated before the read and write functions take in an offset that specifies where to read or write within the BAR. The offset is cumulative and works in the same way as the relative offset mode of the llseek function. For example, if the current offset is 0x40 and a write is issued with an offset of 0x05, the data will be written to BAR address 0x45. It is the responsibility of the read and write functions to update this offset value to account for the most recent operation.

3.2.6 Load Scripts

To help ease development, user space scripts were written to take care of compiling and loading the driver as well as other configuration tasks. The first of these scripts is named device_refresh.sh and allows the FPGA board to be used after reprogramming without needing to power cycle the host system. It accomplishes this task by first determining the PCI ID of the LiteFury using the lspci command. The PCI ID can be used to find the file in the /sys/bus/pci/devices directory that corresponds to the Litefury board. A value of 1 is then echoed to the file at path /sys/bus/pci/devices/0000:$PCI_ID/remove which will prompt the kernel to remove the device. The board can be reacquired by the kernel by echoing 1 to the file at path /sys/bus/pci/rescan, which causes the kernel to rediscover the Litefury board with its newly programmed firmware. This script must be called after every time that the FPGA on the Litefury board is reprogrammed.

The second script, which itself calls the device_refresh.sh script internally, builds and loads the device driver, and also creates device files for the driver in the /dev directory. Building the driver is achieved by calling the drivers makefile. To load, the driver is first removed from the kernel, if it is already loaded, using the rmmod command and then re-inserts the newly built driver using the insmod command. In

order for the driver's device file to be created in the /dev directory, the mknod command is used. For this to work, however, the character device major number must first be acquired from the /proc/devices file and then passed to the mknod command. This script should then be run whenever changes are made to the driver's source code so that these changes can be built and loaded into the kernel.

3.2.7 User Space Library

It is possible for a user space program to directly communicate with the driver, and in turn, with the device, by simply using system calls to interact with the devices file in the /def directory. However, doing so requires that the user has in-depth knowledge of both the device and its firmware. In order to make interacting with the device as seamless as possible, a user space library was created to abstract away the low-level details. The API provides functions for reading, writing, and clearing device registers on the device. However, none of these functions have knowledge of each register's meaning and functionality. As a result, it is up to the user to know what operations, such as reading or writing, are permitted for each register. Additionally, the user must know the offset to each register when using these low-level functions. To simplify this, an additional header named device_specific.h is provided that, among other things, stores the register offsets as easy-to-use macros.

3.3 Prime Finder Project

In order to demonstrate the capability of the hardware and software, beyond basic LED control, an FPGA-based prime number finder was developed to be run on the LiteFury board. This design is provided with a starting number X by the host system and will then find the next prime number that exists after X. Like the LED Control design, this design is divided into an AXI Bus front-end, and a back-end that in this case handles the necessary computation to find prime numbers. To facilitate communication between the user and the hardware, both a device driver as well as a user space libraries and programs were written.

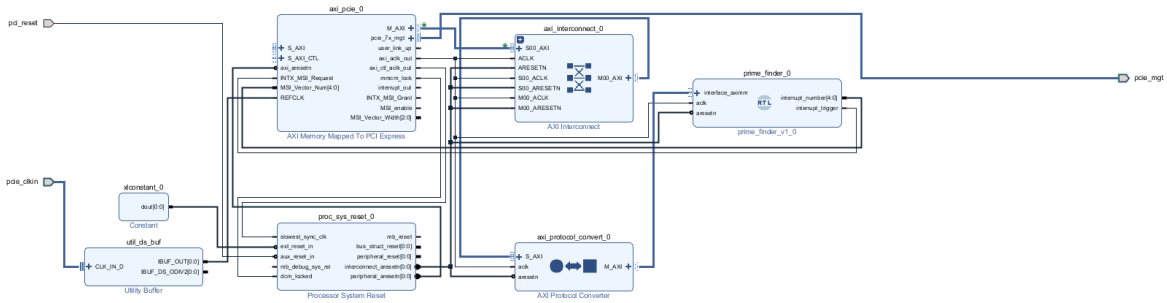

Figure 3.4: Simplified block design for the Prime Finder project

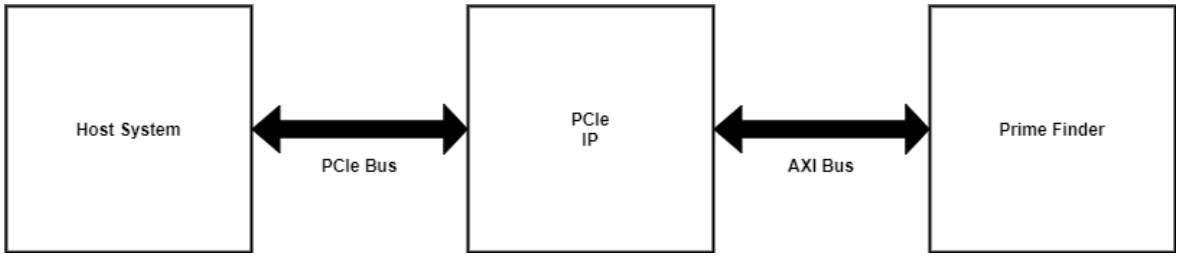

Figure 3.5: Block design from the Vivado IP Integrator for the Prime Finder project

Before continuing, it is useful to state the definition of a prime number. Prime numbers are natural numbers greater than or equal to two that are only evenly divisible by one and itself. The first ten prime numbers are as follows. 2, 3, 5, 7, 11, 13, 17, 19, 23, 29

3.4 Prime Finder FPGA Design Front-End

The front-end of the prime finder design is very similar to that of the LED control design, described in section 3.2.1, with a few key exceptions. While the simplified front-end described in section 3.2.1 has no ability to raise an interrupt on the host system, the prime finder can raise interrupts and includes additional logic to manage this. The second change relates to the number of registers that the front-end contains. Due to the simplicity of the LED Control project, only a single register is required since

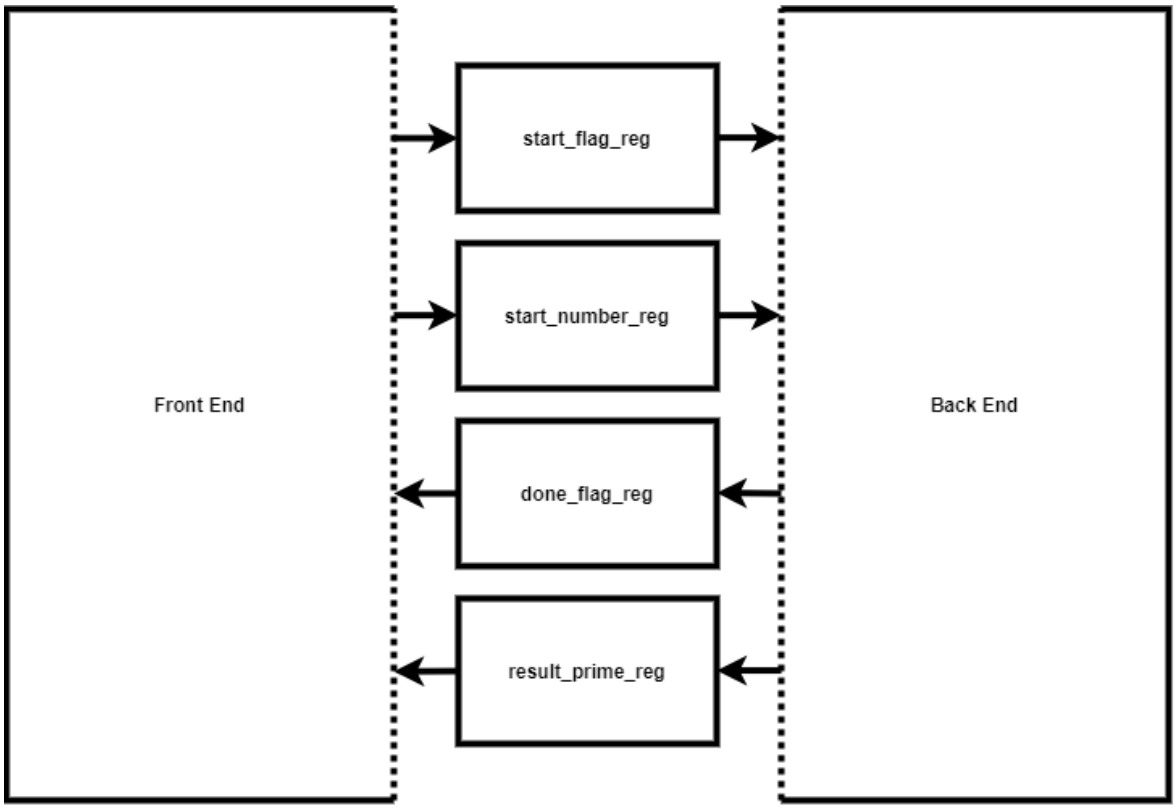

Figure 3.6: Prime-Finder Front-end and back-end communication registers.

each LED only needs a single-bit control signal. This register is bidirectional, meaning that the register is written to control the LED states and read from to acquire the current LED states. The Prime Finder is much more complex and requires that four total registers. Two of these registers are read-only while the other two are write-only, unlike before where the register was bidirectional. The relationship between the front-end and back-end of the prime finder design is shown in figure 3.6.

The registers in the prime finder front-end can be accessed over the AXI bus. Each of these registers is 32bits in width and will be described in detail below:

- START_FLAG
  - Register offset = 0x00
  - When the value in this register transitions from 0 to 1 a new prime number search is started.
- START_NUMBER
  - Register offset = 0x04
  - The value stored in this register is used as the starting value for the prime number search. For example if this register holds a value of 20 the search will start at 21 and search upward.
- DONE_FLAG
  - Register offset = 0x08
  - This register will be set to 1 when a search is completed. While a search is in progress, this register is set to 0. It is therefore possible to detect the completion of a search by polling this registers and detect when it transitions from a zero state to a one state.
- PRIME_NUMBER
  - Register offset = 0x0C
  - Upon completion of the search the prime number that was found will be stored in this register.
- CYCLE_COUNT_HIGH
  - Register offset = 0x10
  - This register stores the upper 32bits of the cycle count from the most recent search operation.

- CYCLE_COUNT_LOW
  - Register offset = 0x14
  - This register stores the lower 32bits of the cycle count from the most recent search operation.

3.4.1 FPGA Prime Design Back-End

The back-end of the prime finder block is made up of a series of nested state machines. The top-level state machine is responsible for cycling through numbers to be checked for primality and reports the discovered prime to the front-end upon the completion of the search, as shown in figure 3.7. To accomplish this goal, it will test each value of N starting with (START_VALUE + 1) by sending it to the prime check state machine and will wait for the results to be delivered. If the prime checker says that N is prime, then the search is complete, and the value of N will be passed back to the front end. If, however, it is determined that N is not prime, N will be incremented, and the process will repeat.

The prime checker is provided with a number N and is tasked with determining its primality. It only works on a single number at a time and has no knowledge of the larger search that is being performed. It is only aware of a single number at any given time and works to determine if that number is a prime. It makes use of the modulus state machine to determine the number's primality. It does this by taking the modulus of N by every value in the range [2, N-1].

$$N \ mod \ i, \ For \ i \ in \ range \ [2..N-1] \tag{3.1}$$

All that must be checked is if the result of the modulus operation is equal to zero. Based on the definition of a prime number, if a given number N is evenly divisible by any number besides itself and one then it is known that N cannot be prime. In this case the prime checker state machine, figure 3.8, will report to the top-level state machine that the number being tested cannot be prime. If the modulus is not equal to zero the primality cannot be determined and more testing must be done. If the end of

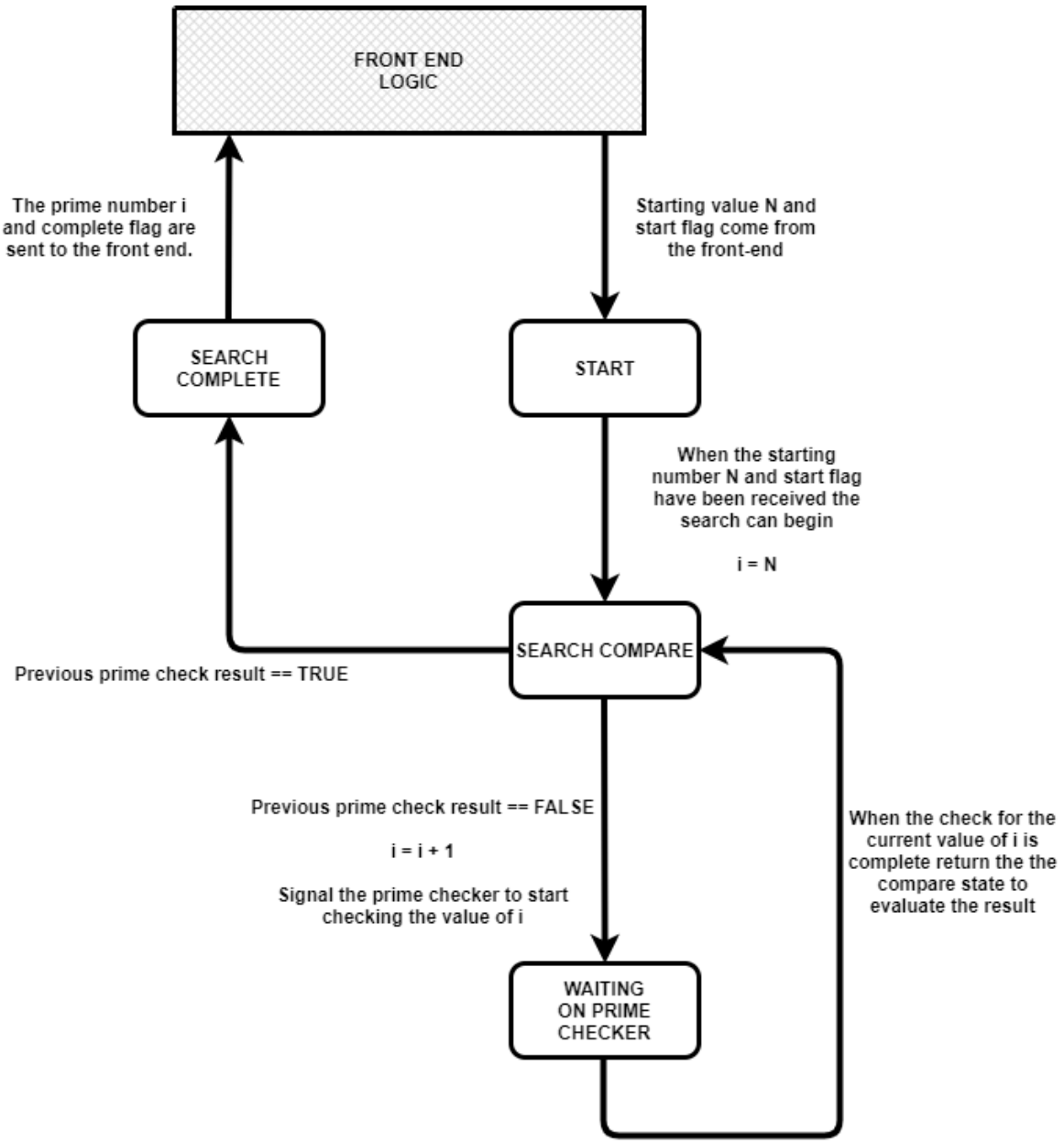

Figure 3.7: State machine diagram for the prime finder back-end.

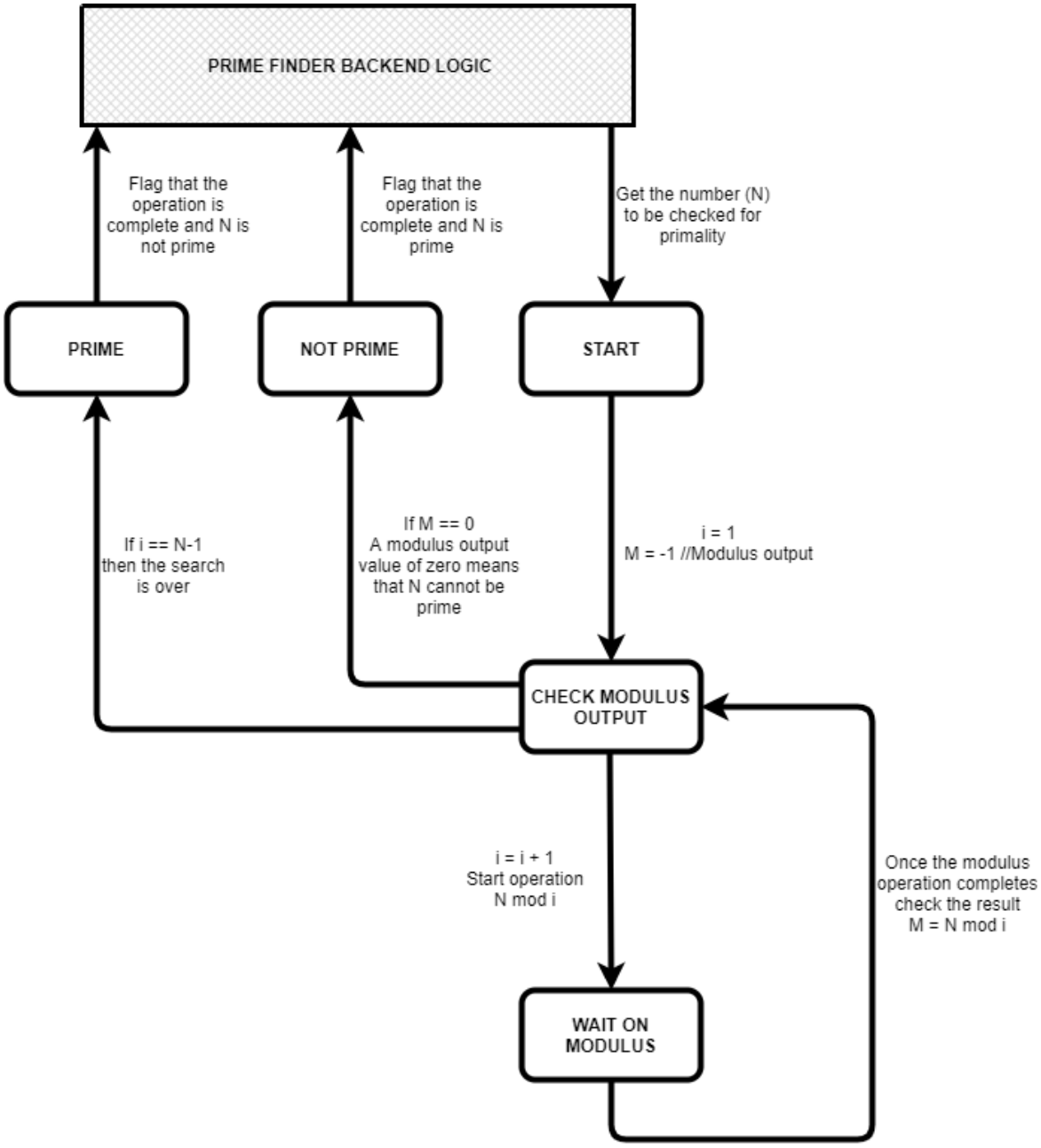

Figure 3.8: State machine diagram for the prime checker.

the range tested, all values up to and including N-1, in equation 1 is reached with no zero results being returned, then the number N is known to be prime, and this result can be provided to the prime finder state machine. For simplicity, the range [2, N-1] is used; however, a more optimized test would only need to search the range [2, $\sqrt{N}$]

While advanced FPGAs such as the Atrix 7 are capable of performing division operations, which in turn allow the modulus to be computed. However, division operations take a significant amount of time to complete and therefore require that the logic be run on a slower clock. In order to allow a higher frequency clock signal to be used without encountering timing errors, a separate state machine was created for this operation that breaks the complicated division operation into smaller chunks that can be completed in a shorter period of time. Each of these smaller operations involves repeatedly subtracting the denominator from the numerator. After each subtraction, a check is performed. If the numerator is now equal to zero, then it is known that the numerator is evenly divisible by the denominator. If, however, the numerator is now less than the denominator, meaning that the next subtraction operation would produce a negative number, then it is known that the numerator is not divisible by denominator. The pseudocode for this operation can be seen in algorithm 1

3.5 Prime Finder Driver

The basic functions of the Prime Finder driver are the same as the LED control driver with more advanced features have been added. These include additional file operations being added along with support for interrupts. All aspects not discussed below can be assumed to match with what was previously discussed in section 3.2.3.

3.5.1 Additional File Operations

The prime finder driver implements all of the file operations previously discussed for the LED control driver along with the additions of the mmap and ioctl functions.

Algorithm 1 Modulus algorithm

```
N ← NUMERATOR
IS_DIVISIBLE ← NULL

loop

  if N == 0 then
    IS_DIVISIBLE ← true
    BREAK
  end if

  if N < DENOMINATOR then
    IS_DIVISIBLE ← false
    BREAK
  end if

end loop
```

The mmap function provides a way to map the device's memory region into a user space process's address space. This allows a user space program interacting with the device to bypass the overhead of the read and write functions.

Next, there is the ioctl function. This function acts as a way of implementing functionality that does not fit well into the standard file I/O methodology. When called, both a command ID and argument data are passed into the function. The command ID provides a way of packing additional functionality into a single function. The argument parameter is an unsigned long type that can either hold argument data directly or hold the address of a structure or buffer in user space. In this driver implementation a command ID of zero is the only valid ID and is used to start a prime number search that will block until the LiteFury's FPGA triggers an interrupt. The argument is used as a way of both passing in the starting value for the search and returning the search result back to user space through a two integer data structure. This allows for both data to be returned as well as a status code. To facilitate this the search result is packed into the argument structure that is copied back to the user. The user space program will fill in the starting value and pass the structure to the driver.

The driver will then populate the search result field and copy the struct back to user space where the necessary data can be extracted.

3.5.2 Interrupts

It is possible for PCIe devices to raise interrupts on the host system. In order for the driver to react properly when an interrupt occurs additional configuration is required. The first part of this configuration process takes place in the PCIe probe function by setting the device as a bus master. A device that is a bus master has the ability to raise interrupts. Next, a list of interrupt vectors must be allocated using the pci_alloc_irq_vectors function. These represent all of the various different interrupt vectors that can be raised by the device. For each interrupt vector that will be used, an IRQ number should be acquired using the pci_irq_vector function. Lastly, the IRQ number can be paired with an interrupt handler function using the request_irq function call. Now when the paired interrupt occurs, the kernel will jump to the handler function.

In the case of the prime finder driver, the interrupt handler will be fired when the LiteFury has completed a prime number search. This interrupt handler is specifically used in conjunction with the ioctl function call, which provides a way to perform a prime number search without the need to resort to polling to detect the completion of a search. The ioctl function will begin a prime number search on the LiteFury board and then wait on a completion structure. Completions provide a way for one process to wait until another task is complete before continuing [3, p. 115]. When the interrupt handler is fired the completion structure will be marked as complete, indicating that the prime number search has finished, and the ioctl function will be allowed to continue.

3.5.3 User Space Library

In order to make the prime finder FPGA design easier to work with and hide the hardware-specific aspects, a user space library was created. The library hides all of the complexity, such as which memory locations to access, when communicating directly

with the hardware. There are two APIs within the library, one for low-level access, which is identical to the API used by the LED control driver, and one for higher-level functionality.

The high-level API provides very specific functions that directly map to device functionality. The first function, start_search, starts a prime number search and will return immediately. Once the search has been started the check_complete function provides a method of polling the DONE_FLAG register on the device to detect the end of the search. When the search is completed the result can be read using the read_result function. Additionally, performance data can be gathered by reading back the number of cycles required for the search from the devices hardware counter registers using read_cycle_count. Lastly, the find_prime function provides a method of running a prime number search without the need to poll any registers. Internally it uses the ioctl function, which blocks waiting for a hardware interrupt to signal the completion of the prime number search.

3.5.4 Performance Data

While performance was not the primary goal of this project, it was none the less possible to collect data through the use of the hardware counters in the FPGA firmware.

The X-axis of figure 3.9 is the number that the prime search started from. The Y-axis is the distance from the start value to the next prime number ($PRIME\ NUMBER - START\ VALUE$). For example, if the start value was 33, then the next prime would be 37. Therefore the distance from 33 to 37 would be 4. Lastly, the Z-axis gives the number of clock cycles required for the prime number search to complete. These are clock cycles of the 125 MHz AXI bus clock.

From the graph data two observations can be made. First, as the start value increases, so does the number of cycles required to complete the search. This is due to the prime checker state machine needing to check all values in the range [2, N-1] to determine the primality of N. Therefore, larger values of N will widen this range and

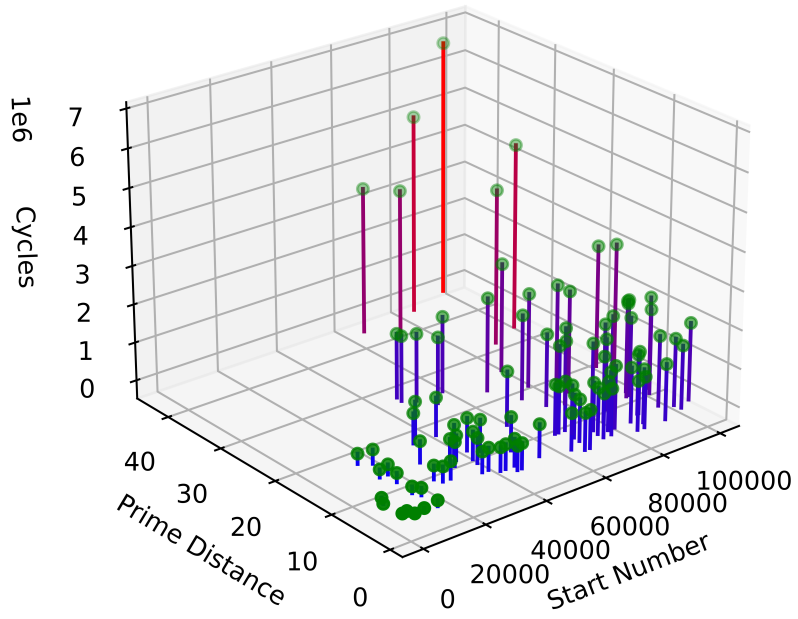

Figure 3.9: Scatter plot of collect data.

require more iterations. The second observation is that the larger the prime distance, the more cycles are required since prime finder state machine will have to check more values before discovering the next prime. The performance of the design is deterministic and does not vary between runs.

Chapter 4

CONCLUSIONS

The work described in this thesis covers the steps required to interface a FPGA accelerator with a Linux host system. The two primary areas of word are the firmware and driver development. For the purposes of this thesis a firmware capable computing prime numbers was developed with a driver developed in tandem to make the accelerator card usable from the Linux PC.

4.1 Conclusion

The architecture of modern computing machines relies heavily on peripheral devices. In order to communicate with these devices, a high-speed interconnect is needed to ensure there is sufficient bandwidth for each peripheral. One of the most common of these is the Peripheral Component Interconnect Express (PCIe). In this thesis, an attempt has been made to document the use of this interface, specifically aimed at the research and education communities.

The primary output product of this research has been the comprehensive sample code and example projects that demonstrate for future users the necessary steps to build a functioning PCIe design complete with drivers, firmware, and user space code. In addition to the sample projects provided, this report provides a detailed description of how each component of the designs functions, in an attempt to provide more information about some of the common stumbling blocks.

The first sample project provides a simple interface by which a user can control the state of the FPGA board's LEDs. To accomplish this goal, implementation of the Linux driver, FPGA firmware, and user space code are provided. The driver allows for data transfer between the host system and the FPGA card by providing a character

device interface to user space programs, and using the Linux Kernel's PCI subsystem to interface with the FPGA. The job of the firmware is to receive the PCIe data from the host, which is converted to AXI4 packets, and based on the received value set the state of the on board LEDs.

The more advanced example project focuses on using the FPGA board to compute prime numbers. This project is in many ways a superset of the LED control project as it implements all of the same high-level functionality but adds the complexity of interrupt handling and the prime finder state machine. Where the LED control driver exclusively used a register communication model this project also allows for the device to generate an interrupt to signal the completion of an operation.

Future work would focus on allowing the device itself to be the master device in a DMA transfer operation. Currently, the device can be treated as a slave device in the DMA transfer operation since it is possible to obtain a physical address to the device's memory. However, there is currently no demonstration as part of this report of the device acting as the DMA master. This functionality is extremely important for any device that captures data and then transfers it to the host system in an efficient manner.

The hope is that this research will be used as a stepping stone for more advanced and complex designs. An area where this research could be especially useful is software defined radio (SDR), in which much of the signal processing can be offloaded to an FPGA-based peripheral. Additionally, such devices can enable the creation of high speed interfaces to external hardware by transferring data to the FPGA to be transmitted. An example of this type of hardware can be seen in modern smart NICs. It is our desire that this thesis will allow future users, especially students and researchers, to quickly get up and running with their projects, and as a result, reduce the barrier to entry for this type of research.

BIBLIOGRAPHY

[1] ARM. Amba® axi™ and ace™ protocol specification.

[2] ARM. Introduction to amba axi.

[3] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers: Where the Kernel Meets the Hardware*. O'Reilly Media, 2005.

[4] Intel. Intel® pci and pci express*.

[5] Intel. Reducing interrupt latency through the use of message signaled interrupts.

[6] kernel.org. Linux kernel licensing rules.

[7] B.W. Kernighan. *Unix: A History and a Memoir*. Independently Published, 2019.

[8] M. J. Koop, W. Huang, K. Gopalakrishnan, and D. K. Panda. Performance analysis and evaluation of pcie 2.0 and quad-data rate infiniband. In *2008 16th IEEE Symposium on High Performance Interconnects*, pages 85–92, 2008.

[9] M. Mares and G. Grundler. *How To Write Linux PCI Drivers*.

[10] GNU Project. Gnu general public license, version 2.

[11] GNU Project. Gnu software.

[12] E.S. Raymond. *The Art of UNIX Programming*. Addison-Wesley Professional Computing Series. Pearson Education, 2003.

[13] Richard Stallman. Linux and the gnu system.

[14] L. Torvalds and D. Diamond. *Just for Fun: The Story of an Accidental Revolutionary*. HarperCollins, 2002.

[15] Xilinx. Amba axi4 interface protocol.

[16] Xilinx. Axi memory mapped to pci express (pcie) gen2 v2.8 logicore ip product guide.

[17] Xilinx. Pci express and xilinx technology.

[18] Xilinx. Vivado design suite axi reference guide.

Appendix A

HELPER SCRIPTS

A.1 Device Refresh Script (device_refresh.sh)

```
#!/bin/bash

PCI_ID=$(lspci | grep -i xilinx | awk '{print $1}')

echo $PCI_ID

echo /sys/bus/pci/devices/0000\:$PCI_ID/remove

echo 1 > /sys/bus/pci/devices/0000\:$PCI_ID/remove
echo 1 > /sys/bus/pci/rescan
```

A.2 Driver Build and Load Script (driver_build_load.sh)

```
#!/bin/bash

DRIVER_NAME=prime_finder
DEVICE_FILE_NAME=prime_finder

./device_refresh.sh

#Remove the driver if it is already loaded
rmmod $DRIVER_NAME
#Rebuild the driver
make
#Reload the driver once the build has finished
insmod $DRIVER_NAME.ko

MAJOR_NUMBER=‘cat /proc/devices | grep $DRIVER_NAME | awk '{print $1}'‘

#The c argument creates a character device
mknod /dev/$DEVICE_FILE_NAME c $MAJOR_NUMBER 0
```

Appendix B

LINUX DRIVER CODE

B.1 Main file (prime_finder_main.c)

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/cdev.h>
#include <linux/semaphore.h>
#include <linux/vmalloc.h>
#include <asm/byteorder.h>
#include <linux/mm.h>

//Defines macros for each register in the prime finder device
#include "file_ops.h"
#include "pcie_ctrl.h"

//Device major and minor numbers
dev_t char_device_numbers;

//Character device representation within the kernel.
struct cdev char_device;

//This variable tracks what setup steps have been completed
//so that these steps can be undone in case of an error or
//when the driver is unloaded
unsigned int setup_status;

void back_out_char_device(void) {

    //Runs through the steps in reverse order that they
    //were done during setup.
    switch(setup_status) {
        case 3:
            pci_unregister_driver(&pci_driver_struct);
        case 2:
```

```
            cdev_del(&char_device);
        case 1:
            unregister_chrdev_region(char_device_numbers, 1);
    }

}

//Function for seting up the driver
static int __init startup(void) {
    int err;

    //Setup status is declared above and is used to
    //track setup steps so that they can be undone
    //later
    setup_status = 0;

    printk(KERN_INFO "Startup\n");

    //Get major and minor numbers for the charater device
    err = alloc_chrdev_region(&char_device_numbers, 0, 1, DEVICE_NAME);
    if(err < 0) {
        printk(KERN_WARNING "Falid to allocate defice numbers\n");
        back_out_char_device();
        return -1;
    }
    setup_status++;

    //Register the character device
    cdev_init(&char_device, &file_ops);

    //Once the character device is added it is
    //considered to be live
    err = cdev_add(&char_device, char_device_numbers, 1);
    if(err < 0) {
        printk(KERN_WARNING "Failed to add the character device\n");
        back_out_char_device();
        return -1;
    }
    setup_status++;

    //Register this driver with the PCI subsystem.
    err = pci_register_driver(&pci_driver_struct);
    if(err < 0) {
        printk("Failed to register PCI device\n");
```

```
        back_out_char_device();
        return -1;
    }
    setup_status++;

    printk(KERN_INFO "Startup Complete\n");

    return 0;
}

static void __exit shutdown(void) {
    printk(KERN_INFO "Shutdown\n");
    //This function will backout the setup steps in
    //the reverse order that they occurred.
    back_out_char_device();
    printk(KERN_INFO "Shutdown Complete\n");
}

module_init(startup);
module_exit(shutdown);

MODULE_LICENSE("MIT");
\end{verbatim}
}

\section{Device Specific Values (device\_specific.h)}
{\fontfamily{ptm}\selectfont
\begin{verbatim}
#define LITEFURY_VENDOR_ID 0x10EE
#define LITEFURY_DEVICE_ID 0x7014

#define DEVICE_NAME "prime_finder"


//Register offsets
#define START_FLAG 0
#define START_NUMBER 4
#define DONE_FLAG 8
#define PRIME_NUMBER 12
#define CYCLE_COUNT_HIGH 16
#define CYCLE_COUNT_LOW 20

\end{verbatim}
}
```

```
\section{File Operations Header (file_ops.h)}
{\fontfamily{ptm}\selectfont
\begin{verbatim}
#ifndef FILE_OPS_H
#define FILE_OPS_H

#include <linux/fs.h>
#include <linux/completion.h>

/*
    Allows the userspace program to map BAR0 into
    its address space

    Parameters:
        filep   -> Pointer to the devices file
                   structure.
        vma     -> Structure pointer describing
                   the user space processes
                   virtual address region to map
                   BAR0 into.

    Return:
        0 on success and a negative value otherwise.
*/
int mmap(struct file *filep, struct vm_area_struct *vma);

/*
    Performs a blocking read from the device's BAR0.
    NOTE: For simplicity this function only reads
    a single 32bit value regardless of how large the
    provided buffer is.

    Parameters:
        filep   -> Pointer to the devices file
                   structure.
        buff    -> User space buffer from user
                   space. Cannot be directly
                   accessed in kernel space.
        count   -> Indicates the size of the
                   buffer pointed to by buff.
        offp    -> Offset to read from with
                   in the file. In this
                   case this is the offset
```

```
                    to read from with in BAR0.
    Return:
        Returns the number of bytes read during
        the operation.
*/
ssize_t read(struct file *filp, char __user *buff, size_t count, loff_t *offp);

/*
    Function to write data to the PCIe device's BAR0.
    NOTE: For simplicity this function only writes
    a single 32bit value regardless of how large
    the provided buffer is.
    Parameters:
        filep   -> Pointer to the devices file structure.
        buff    -> User space buffer from user space.
                   Cannot be directly accessed in kernel
                   space.
        count   -> Indicates the size of the buffer pointed
                   to by buff.
        offp    -> Offset to write to from with in the file.
                   In this case this is the offset to
                   write to with in BAR0.
    Return:
        Returns the number of bytes written during the operation.
*/
ssize_t write(struct file *filp, const char __user *buff, size_t count, loff_t *offp);

/*
    Function to let user space programs open the
    driver's device file.
    Parameters:
        inode   -> Pointer to the devices inode sturcture
        filep   -> Pointer to the devices file sturcture
        NOTE: For more detials on these parametrs look
              up the specific
        structure types.
    Return:
        0 on success negative value on failure.
*/
int open (struct inode *inode, struct file *filp);

/*
    Function to let user space programs close the
    driver's device file.
```

```
    Parameters:
        inode   -> Pointer to the devices inode sturcture
        filep   -> Pointer to the devices file sturcture
        NOTE: For more detials on these parametrs look
              up the specific
        structure types.
    Return:
        0 on success negative value on failure.
*/
int release(struct inode *inode, struct file *filp);

/*
    Allows to set the offset that will be written to or
    read from.
    Parameters:
        filep   -> Pointer to the devices file structure.
        offset  -> The value used to set the new offset
        whence  -> Indicates where the offset should be
                   set from
                   The options are SEEK_SET or SEEK_CUR.
                   When SEEK_SET is used the offset is
                   set relative to the start position.
                   When SEEK_CUR is used the offset is
                   set from the current position.
    Return:
        Returns the newly set offset value.
*/
loff_t llseek(struct file *filp, loff_t offset, int whence);

/*
    Function for non-standard I/O and control
    functions. In this driver it is used
    to activate a blocking prime search where
    the function will wait for the FPGA to
    finish its search before returning.

    Parameter:
        filp    -> Pointer to the devices file
                   structure.
        cmd     -> Command ID. Currently the only
                   valid command ID is 0.
        arg     -> Argument value. What this value
                   represents can change based on
                   use case but in this driver it
```

```
                    is a pointer to an ioctl_struct
                    in userspace.

    Return:
        Returns 0 on success and a negative value
        on failure.
*/
long int ioctl(struct file *filp, unsigned int cmd, unsigned long arg);

//This structure holds all of the file operations
//that the driver supports
extern const struct file_operations file_ops;

extern struct completion ioctl_completion;

#endif
```

B.2 File Operations Implementation (file_ops.c)

```
#include "file_ops.h"
#include "device_specific.h"
#include "pcie_ctrl.h"

#include <linux/uaccess.h>
#include <linux/slab.h>
#include <linux/types.h>


const struct file_operations file_ops = {
    .owner = THIS_MODULE,
    .read = read,
    .write = write,
    .llseek = llseek,
    .open = open,
    .release = release,
    .mmap = mmap,
    .unlocked_ioctl = ioctl,
};


DECLARE_COMPLETION(ioctl_completion);


//This struct is defined here since it should not
```

```
//be used outside of this file. This structure is
//mirrored in prime.c but uses the stdint.h
//integer definitions (uint32_t).
struct ioctl_struct {
    u32 start_val;
    u32 search_result;
};


/*
    Function for non-standard I/O and control functions.
    In this driver it is used to activate a blocking
    prime search where the function will wait for the
    FPGA to finish its search before returning.

    Parameter:
        filp    -> Pointer to the devices file structure.
        cmd     -> Command ID. Currently the only valid
                   command ID is 0.
        arg     -> Argument value. What this value represents
                   can change based on use case but in this
                   driver it is a pointer to an ioctl_struct
                   in userspace.

    Return:
        Returns 0 on success and a negative value
        on failure.
*/
long int ioctl(struct file *filp, unsigned int cmd, unsigned long arg) {

    //Case 0 variables
    u32 start_value;
    int completion_status;
    struct ioctl_struct kernel_space_struct;
    unsigned long not_copied_count;
    struct ioctl_struct __user *user_space_ptr;

    //Pring logging data
    printk(KERN_INFO "IOCTL: %d\n", cmd);
    printk(KERN_INFO "%IOCTL ARG: lu\n", arg);


    switch(cmd) {
```

```
//0 -> blocking prime search operation
case 0:

    //The arg is a pointer to a userspace
    //structure containing the start value
    //for the search and an additionally
    //field for returning the result of
    //the search.
    user_space_ptr = (struct ioctl_struct*) arg;

    //Check that the userspace pointer is valid
    if(!access_ok(user_space_ptr, sizeof(struct ioctl_struct))) {
        printk(KERN_INFO "Ioctl struct error\n");
        return -1;
    }

    //Copy the userspace struct to kernel space
    not_copied_count = copy_from_user(&kernel_space_struct, \
                                      user_space_ptr, \
                                      sizeof(struct ioctl_struct));

    //Make sure all of the data could be copied
    if(not_copied_count != 0) {
        printk(KERN_INFO "Failed to copy ioctl struct from user space\n");
        return -2;
    }

    start_value = (u32) kernel_space_struct.start_val;
    //Write the start value
    iowrite32(start_value, bar0_ptr + START_NUMBER);
    //Set the start bit
    iowrite32(1, bar0_ptr + START_FLAG);

    //Wait for the interrupt to fire which tells
    //us the task is complete
    if( wait_for_completion_interruptible(&ioctl_completion) != 0 ) {
        return -3;
    }

    //Read back the value and return the result
    kernel_space_struct.search_result = ioread32(bar0_ptr + PRIME_NUMBER);

    //Copy the structure back to user space
    not_copied_count = copy_to_user(user_space_ptr, \
```

```
                                        &kernel_space_struct, \
                                        sizeof(struct ioctl_struct));

            if(not_copied_count != 0) {
                printk(KERN_INFO "Faild to copy ioctl struct from user space\n");
                return -2;
            }

            return 0;

        default:
            return -1;

    }
}


/*
    Allows the userspace program to map BAR0 into its
    address space

    Parameters:
        filep   -> Pointer to the devices file structure.
        vma     -> Structure pointer describing the user space
                   processes virtual address region to map
                   BAR0 into.

    Return:
        0 on success and a negative value otherwise.
*/
int mmap(struct file *filep, struct vm_area_struct *vma) {
    int status;

    //Convert the page offset to an address offset
    unsigned long off = vma->vm_pgoff << PAGE_SHIFT;

    //The VM_RESERVED flag has been replaced by VM_DONTEXPAND and
    //VM_DONTDUMP in newer kernel versions
    vma->vm_flags = VM_IO | VM_DONTEXPAND | VM_DONTDUMP;

    //Make sure that the memory region is not cached
    vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);

    //Actually perform the mapping. NOTE that the offset parameter
```

```
    //is in terms of pages which is why the >>PAGE_SHIFT is needed
    //inorder to get back to pages from an address.
    status = io_remap_pfn_range(vma, \
                                vma->vm_start, \
                                (bar0_start+off)>>PAGE_SHIFT, \
                                vma->vm_end - vma->vm_start, \
                                vma->vm_page_prot);

    //Log the operation
    printk("MMAP STATUS: %d\n", status);
    printk("MMAP START ADDRESS: %lu\n", vma->vm_start);
    return status;
}


/*
    Performs a blocking read from the device's BAR0. NOTE: For
    simplicity this function only reads a single 32bit value
    regardless of how large the provided buffer is.

    Parameters:
        filep   -> Pointer to the devices file structure.
        buff    -> User space buffer from user space. Cannot be
                   directly accessed in kernel space.
        count   -> Indicates the size of the buffer pointed to by buff.
        offp    -> Offset to read from with in the file. In this case this is
                   the offset to read from with in BAR0.
    Return:
        Returns the number of bytes read during the operation.
*/
ssize_t read(struct file *filp, char __user *buff, size_t count, loff_t *offp) {
    //The way that this works is that the offset is not "remembered" between
    //function calls
    unsigned int val;
    unsigned long not_copied_count;

    //Check that the user space buffer is OK to be used
    if(!access_ok(buff, count)) {
        printk(KERN_INFO "Read buffer error\n");
        return -1;
    }

    //Log the operation in the kernel log
    printk(KERN_INFO "READ\n");
```

```
    printk(KERN_INFO "READ OFFSET: %lld", *offp);

    //Read in the value at the provided offset from the BAR0 start.
    val = ioread32(bar0_ptr + *offp);

    //Move the offset by the amount read. This is stored between
    //operations.
    *offp += count;

    not_copied_count = copy_to_user(buff, &val, sizeof(unsigned int));
    return (count - not_copied_count);
}

/*
    Function to write data to the PCIe device's BAR0. NOTE: For simplicity this function
    only writes a single 32bit value regardless of how large the provided buffer is.
    Paramaters:
        filep   -> Pointer to the devices file sturcture.
        buff    -> User space buffer from user space. Cannont be directly accessed
                   in kernel space.
        count   -> Indicates the size of the buffer pointed to by buff.
        offp    -> Offset to write to from with in the file. In this case this is
                   the offset to write to with in BAR0.
    Return:
        Returns the number of bytes written during the operation.
*/
ssize_t write(struct file *filp, const char __user *buff, size_t count, loff_t *offp) {

    unsigned long not_copied_count;
    u32 *kernel_ptr;

    //Check that the userspace buffer is valid
    if(!access_ok(buff, count)) {
        printk(KERN_INFO "Write buffer error\n");
        return -1;
    }

    //Copy the userspace buffer to a kernel space buffer. This is needed
    //inorder to use the iowrite32 function which needs a kernal space
    //address.
    kernel_ptr = (u32*) kmalloc(count * sizeof(char), GFP_KERNEL);
    not_copied_count = copy_from_user(kernel_ptr, buff, count);

    //Log the operation in the kernel log
```

```
    printk(KERN_INFO "WRITE\n");
    printk(KERN_INFO "WRITE OFFSET: %lld", *offp);

    //Read in the value at the provided offset from the BAR0 start.
    iowrite32(kernel_ptr[0], bar0_ptr + *offp);

    //Free the internal buffer
    kfree(kernel_ptr);

    //Move the offset by the amount read. This is stored between
    //operations.
    *offp += count;

    return (count - not_copied_count);
}


/*
    Function to let user space programs open the driver's device file.
    Paramaters:
        inode   -> Pointer to the devices inode sturcture
        filep   -> Pointer to the devices file sturcture
        NOTE: For more detials on these parametrs look up the specific
        structure types.
    Return:
        0 on success negative value on failure.
*/
int open (struct inode *inode, struct file *filp) {
    //Nothing to be done here. Just log that the file was
    //opened and return.
    printk(KERN_INFO "File Opened\n");

    return 0;
}

/*
    Function to let user space programs close the driver's device file.
    Paramaters:
        inode   -> Pointer to the devices inode sturcture
        filep   -> Pointer to the devices file sturcture
        NOTE: For more detials on these parametrs look up the specific
        structure types.
    Return:
        0 on success negative value on failure.
```

```
*/
int release(struct inode *inode, struct file *filp) {
    //Nothing to be done here. Just log that the file was
    //closed and return.
    printk(KERN_INFO "File Closed\n");

    return 0;
}

/*
    Allows to set the offset that will be written to or read from.
    Paramaters:
        filep   -> Pointer to the devices file sturcture.
        offset  -> The value used to set the new offset
        whence  -> Indicates where the offset should be set from
                   The options are SEEK_SET or SEEK_CUR. When SEEK_SET
                   is used the offset is set relative to the start
                   position. When SEEK_CUR is used the offset is
                   set from the current position.
    Return:
        Returns the newly set offset value.
*/
loff_t llseek(struct file *filp, loff_t offset, int whence) {

    //Set the offset relative to the start (In absolute terms).
    if(whence == SEEK_SET) {
        filp->f_pos = offset;
    }
    //Set the offset relative to the current position
    else if(whence == SEEK_CUR){
        filp->f_pos += offset;
    }

    //Log the operation.
    printk(KERN_INFO "SEEK\n");
    printk(KERN_INFO "SEEK OFFSET: %lld\n", filp->f_pos);

    return filp->f_pos;
}


\end{verbatim}
}
```

```
\section{PCIe Control Header (pcie_ctrl.h)}
{\fontfamily{ptm}\selectfont
\begin{verbatim}
#ifndef PCIE_CTRL_H
#define PCIE_CTRL_H

#include "device_specific.h"

#include <linux/pci.h>
#include <linux/interrupt.h>
#include <linux/module.h>

//Interrupt handler function
static irqreturn_t interrupt_handler(int irq, void *dev);

/*
    This function is called when the kernel finds a device that can be
    paired with the driver.

    Parameters:
        dev     -> Device structure pointer of the device the driver is
                   being paired with.
        id      -> Pointer to the ID information of the device being
                   paired.

    Return:
        0 on success and a negative value on failure.
*/
int pci_probe (struct pci_dev *dev, const struct pci_device_id *id);

/*
    This function is called when the device is removed.
    Paramaters:
        dev     -> Pointer to the device the driver is paired with
    Return:
        Nothing.
*/
void pci_remove (struct pci_dev *dev);

//This array contains the several PCI device id structures. These structures
//have several feilds but in this case only the vendor id and device id are used.
//PCI_DEVICE is a helper macro for initializing a structure instance.
//It is important that this array end with a NULL entry which in this case
//is {0, }
```

```
static struct pci_device_id pci_id_array[] = {
    { PCI_DEVICE(LITEFURY_VENDOR_ID, LITEFURY_DEVICE_ID)},
    { 0, }
};


//Maps various PCI related functions and values into the struct.
//This is then used to register the driver with the PCI subsystem.
//There are additional feilds in the sturcture but this are the
//minimum required feilds.
static struct pci_driver pci_driver_struct = {
    .name = DEVICE_NAME,
    .id_table = pci_id_array,
    .probe = pci_probe,
    .remove = pci_remove
};



//Pointer to the start of the BAR0 address space AFTER it has been
//mapped into the virtual address space.
extern char *bar0_ptr;
//Data related to bar 0 on the device
extern unsigned long bar0_size;
extern unsigned long bar0_start;

extern u8 interrupt_number;

#endif
\end{verbatim}
}

\section{File Operations Implementation (file_ops.c)}
{\fontfamily{ptm}\selectfont
\begin{verbatim}
#include "pcie_ctrl.h"
#include "file_ops.h"

//Add data about supported devices to the module table so the kernel
//knows what devices this drives should be paired with.
MODULE_DEVICE_TABLE(pci, pci_id_array);


//Pointer to the start of the BAR0 address space AFTER it has been
```

```
//mapped into the virtual address space.
char *bar0_ptr;
//Data related to bar 0 on the device
unsigned long bar0_size;
unsigned long bar0_start;

u8 interrupt_number;

//Interrupt handler function
static irqreturn_t interrupt_handler(int irq, void *dev) {
    printk(KERN_INFO "INTERRUPT: %d\n", irq);
    complete(&ioctl_completion);
    return IRQ_HANDLED;
}

/*
    This function is called when the kernel finds a device that can be
    paired with the driver.

    Parameters:
        dev     -> Device structure pointer of the device the driver is
                   being paired with.
        id      -> Pointer to the ID information of the device being
                   paired.

    Return:
        0 on success and a negative value on failure.
*/
int pci_probe (struct pci_dev *dev, const struct pci_device_id *id) {
    int status;
    u16 vendor_id;

    //Store the addres of both the start and end of the PCIe memory region
    unsigned long bar0_ptr_int_start;
    unsigned long bar0_ptr_int_end;

    printk(KERN_INFO "PCI PROBE\n");
    status = pci_enable_device(dev);

    if(status != 0) {
        return status;
    }

    //Read the vendor ID from the configuration space of the device.
```

```
    status = pci_read_config_word(dev, PCI_VENDOR_ID, &vendor_id);

    if(status != 0) {
        return status;
    }

    //All PCI values are big endian so the conversion to the cpu byte ordering
    //is required to make sure this works on all platforms.
    printk(KERN_INFO "%d\n", be16_to_cpu(vendor_id));

    //Get the start and end addresses of the devices BAR0 memory region.
    bar0_ptr_int_start = pci_resource_start(dev, 0);
    bar0_ptr_int_end = pci_resource_end(dev, 0);

    bar0_size = bar0_ptr_int_end - bar0_ptr_int_start;
    bar0_start = bar0_ptr_int_start;

    //Map the BAR0 memory region of the device into the virtual address space.
    bar0_ptr = (char*) ioremap(bar0_ptr_int_start, bar0_size);

    //Make the device a bus master so that it can raise interrupts
    pci_set_master(dev);

    //Allocate a single interrupt vector
    int vector_count = pci_alloc_irq_vectors(dev, 1, 1, PCI_IRQ_MSI);
    printk(KERN_INFO "Allocated Vector Count: %d\n", vector_count);

    //Get the IRQ number for the vector
    interrupt_number = pci_irq_vector(dev, 0);
    printk(KERN_INFO "Assigned IRQ: %d\n", interrupt_number);

    //Attach a handler to the IRQ number
    int irq_request_status = request_irq(interrupt_number, \
                                         interrupt_handler, \
                                         IRQF_SHARED, \
                                         DEVICE_NAME, \
                                         dev);
    printk(KERN_INFO "IRQ Request Status: %d\n", irq_request_status);

    return status;
}

/*
    This function is called when the device is removed.
```

```
    Paramaters:
        dev     -> Pointer to the device the driver is paired with
    Return:
        Nothing.
*/
void pci_remove (struct pci_dev *dev) {

    //Free up the interrupt
    free_irq(interrupt_number, dev);
    //Free up the interrupt vectors
    pci_free_irq_vectors(dev);
    //Un-map BAR0 from kernel space
    iounmap(bar0_ptr);
    //Disable the device
    pci_disable_device(dev);
    printk(KERN_INFO "PCI REMOVE\n");
}
```

Appendix C

USER SPACE CODE

C.1 Prime Finder Control Library Header (prime.h)

```
////////////////////////////////////////////////////
//Low-level API
////////////////////////////////////////////////////
int clear_registers(int fd);
int read_register(int fd, int reg_offset, uint32_t *value);
int write_register(int fd, int reg_offset, uint32_t value);

////////////////////////////////////////////////////
//High-level API
////////////////////////////////////////////////////

/*
    Starts a search by writting to the start search register
    on the device.

    Paramaters:
        fd              -> File descriptor of the drivers device file.
        start_val       -> Value to start the prime search from.
    Return:
        On success zero is returned, on failure a negative
        value is returned.
*/
int start_search(int fd, uint32_t start_val);

/*
    Checks if a previously startes search has completed.

    Paramaters:
        fd              -> File descriptor of the drivers device file.
        search_status   -> Pointer to where the result of the query
                           should be stored.
```

```
    Return:
        On success zero is returned, on failure a negative
        value is returned.
*/
int check_complete(int fd, uint32_t *search_status);

/*
    Reads the result of the prime number search from the devices
    result register.

    Paramaters:
        fd              -> File descriptor of the drivers device file.
        result          -> Pointer to where the result of the query
                           should be stored.

    Return:
        On success zero is returned, on failure a negative
        value is returned.
*/
int read_result(int fd, uint32_t *result);

/*
    Reads the number of cycles taken to complete the previous prime
    number search.

    Paramaters:
        fd              -> File descriptor of the drivers device file.
        cycles          -> Pointer to where the result of the query
                           should be stored.

    Return:
        On success zero is returned, on failure a negative
        value is returned.
*/
int read_cycle_count(int fd, uint64_t *cycles);

/*
    Starts a blocking prime search where the search completion will be
    signaled by an interrupt.

    Paramaters:
        fd              -> File descriptor of the drivers device file.
        search_result   -> Pointer to where the result of the search
                           should be stored.
```

```
    Return:
        On success zero is returned, on failure a negative
        value is returned.
*/
int find_prime(int fd, uint32_t start_val, uint32_t *search_result);
\end{verbatim}
}

\section{Prime Finder Control Library Header (prime.c)}
{\fontfamily{ptm}\selectfont
\begin{verbatim}
#include <stdint.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "device_specific.h"

////////////////////////////////////////////////////
//Low-level API
////////////////////////////////////////////////////

/*
    Write zero to each of the user writable registers on the device.

    Parameters:
        fd  -> File descriptor of the device file.

    Return:
        0 on success and a negative value otherwise.
*/
int clear_registers(int fd) {
    uint32_t data = 0;
    int status;

    //Move the file pointer to the start flag register then write zero to it
    //and check for errors.
    lseek(fd, START_FLAG, SEEK_SET);
    status = write(fd, &data, sizeof(data));
    if(status == -1) return -1;

    //There is no need to seek to the next register value since the previous write
    //operation will already have moved the file pointer
    status = write(fd, &data, sizeof(data));
```

```
    if(status == -1) return -1;

    return 0;
}

/*
    Reads the value of the register at a given offset.
    Paramaters:
        fd          -> File descriptor of the device file.
        reg_offset  -> Offset of the register to be read.
        value       -> Pointer to where the value read from
                       the register should be stored.
    Return:
        0 on success and a negative value otherwise.
*/
int read_register(int fd, int reg_offset, uint32_t *value) {
    int read_count;

    //Move to the offset of the register
    lseek(fd, reg_offset, SEEK_SET);

    //Read the value
    read_count = read(fd, value, sizeof(uint32_t));
    //Check that the correct amount of data was read. (32 bit == 4 bytes)
    if(read_count != 4) return -1;

    return 0;
}

/*
    Writes a given value to the register at a given offset.
    Paramaters:
        fd          -> File descriptor of the device file.
        reg_offset  -> Offset of the register to be written to.
        value       -> Value to be written to the register.
    Return:
        0 on success and a negative value otherwise.
*/
int write_register(int fd, int reg_offset, uint32_t value) {

    int write_count;

    //Move to the correct register offset
    lseek(fd, reg_offset, SEEK_SET);
```

```
    //Write the value
    write_count = write(fd, &value, sizeof(uint32_t));

    //Check that the correct amount of data was written. (32 bit == 4 bytes)
    if(write_count != 4) return -1;
    else return 0;

}

////////////////////////////////////////////////////
//High-level API
////////////////////////////////////////////////////


/*
    Starts a search by writting to the start search register
    on the device.

    Paramaters:
        fd              -> File descriptor of the drivers device file.
        start_val       -> Value to start the prime search from.
    Return:
        On success zero is returned, on failure a negative
        value is returned.
*/
int start_search(int fd, uint32_t start_val) {
    const uint32_t start_flag = 1;
    int status;

    status = write_register(fd, START_NUMBER, start_val);
    if(status == -1) return -1;
    status = write_register(fd, START_FLAG, start_flag);
    if(status == -1) return -1;

    return 0;
}

/*
    Checks if a previously startes search has completed.

    Paramaters:
        fd              -> File descriptor of the drivers device file.
        search_status   -> Pointer to where the result of the query
                           should be stored.
```

```
    Return:
        On success zero is returned, on failure a negative
        value is returned.
*/
int check_complete(int fd, uint32_t *search_status) {
    uint32_t flag_register_val;
    int status = read_register(fd, DONE_FLAG, &flag_register_val);

    if(status != 0) {
        return -1;
    }

    if(flag_register_val == 1) {
        *search_status = 1;
    }
    else {
        *search_status = 0;
    }

    return 0;
}

/*
    Reads the result of the prime number search from the devices
    result register.

    Paramaters:
        fd              -> File descriptor of the drivers device file.
        result          -> Pointer to where the result of the query
                           should be stored.

    Return:
        On success zero is returned, on failure a negative
        value is returned.
*/
int read_result(int fd, uint32_t *result) {
    return read_register(fd, PRIME_NUMBER, result);
}


/*
    Reads the number of cycles taken to complete the previous prime
    number search.
```

```
    Paramaters:
        fd              -> File descriptor of the drivers device file.
        cycles          -> Pointer to where the result of the query
                           should be stored.

    Return:
        On success zero is returned, on failure a negative
        value is returned.
*/
int read_cycle_count(int fd, uint64_t *cycles) {
    uint32_t upper_bits = 0, lower_bits = 0;
    int status = 0;

    status = read_register(fd, CYCLE_COUNT_HIGH, &upper_bits);
    if(status != 0) {
        return -1;
    }

    status = read_register(fd, CYCLE_COUNT_LOW, &lower_bits);
    if(status != 0) {
        return -1;
    }

    //Combine the upper and lower register values
    *cycles = ( ((uint64_t)upper_bits << 32) | lower_bits );

    return 0;
}

//This scruct is defined here since it should not be used outside
//of this file. This structure is mirrored in file_ops.c but uses
//the kernels internal integer definitions (u32).
struct ioctl_struct {
    uint32_t start_val;
    uint32_t search_result;
};

/*
    Starts a blocking prime search where the search completion will be
    signaled by an interrupt.

    Paramaters:
        fd              -> File descriptor of the drivers device file.
```

```
        search_result   -> Pointer to where the result of the search
                           should be stored.
    Return:
        On success zero is returned, on failure a negative
        value is returned.
*/
int find_prime(int fd, uint32_t start_val, uint32_t *search_result) {
    int status;

    //Fill in the start value field of the structure
    struct ioctl_struct user_space_struct;
    user_space_struct.start_val = start_val;

    //This function will block until the device raises an
    //interrupt to indicate the search is complete.
    status = ioctl(fd, 0, &user_space_struct);

    if(status == 0) {
        //Retreive the search result from the structure.
        *search_result = user_space_struct.search_result;
        return 0;
    }
    else {
        return -1;
    }
}
\end{verbatim}
}

\section{User Space Test Program (user_space_test.c)}
{\fontfamily{ptm}\selectfont
\begin{verbatim}
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdlib.h>
#include <sys/ioctl.h>

#include "prime.h"


int main(int argc, char *argv[]) {
    int count;
```

```
//Open the device file and check that it was opened correctly
int fd = open("/dev/prime_finder", O_RDWR);
if(fd < 0) {
    printf("Failed to open device file\n");
    return -1;
}


//Determine the number that the prime number search should start from
//If a start number was provided on the command line then use that
unsigned int start_number;
if(argc >= 2) {
    start_number = atol(argv[1]);
}
//Otherwise ask the user to provide one
else {
    printf("Enter the start number: ");
    scanf("%ud", &start_number);
}

//Clear all of the registers on the device and then start the prime number search
clear_registers(fd);

////////////////////////////////////////////////////////////////////////////
//To run the test using polling uncomment this code and commend the below code
////////////////////////////////////////////////////////////////////////////

int status;

status = start_search(fd, start_number);
if(status != 0) {
    printf("Error starting search\n");
    return -1;
}

//Busy loop until the prime search completes
uint32_t complete;
do {
    status = check_complete(fd, &complete);
    if(status != 0) {
        printf("Error checking search completion\n");
        return -1;
    }
```

```
        usleep(250000);
    } while(complete != 1);

    uint64_t cycle_count;
    status = read_cycle_count(fd, &cycle_count);
    if(status != 0) {
        printf("Error reading cycle count\n");
        return -1;
    }
    printf("Cycle count: %lu\n", cycle_count);

    uint32_t result;
    status = read_result(fd, &result);
    if(status != 0) {
        printf("Error reading search result\n");
        return -1;
    }
    printf("Prime search result: %u\n", result);


    ////////////////////////////////////////////////
    //Test using blocking
    ////////////////////////////////////////////////

    // uint32_t prime;
    // find_prime(fd, start_number, &prime);
    // printf("%d\n", prime);

    return 0;
}
```