2018-06-01

High Throughput FPGA Configuration Using a Custom DMA Configuration Controller

Peter William Zabriskie
Brigham Young University

High Throughput FPGA Configuration Using a Custom DMA Configuration Controller

Peter William Zabriskie

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Michael J. Wirthlin, Chair
Brent E. Nelson
Jeffrey Goeders

Department of Electrical and Computer Engineering

Brigham Young University

ABSTRACT

High Throughput FPGA Configuration Using a Custom DMA Configuration Controller

Peter William Zabriskie
Department of Electrical and Computer Engineering, BYU
Master of Science

SRAM-based Field Programmable Gate Arrays (FPGAs) must be programmed with configuration data every time they are powered on. In addition to initially programming an FPGA, there are many other applications that require access to FPGA configuration memory such as partial reconfiguration, fault injection, and memory scrubbing. This thesis describes a system that provides high-speed, programmable configuration management for Xilinx FPGAs through external interfaces. This system is an improvement upon the JTAG Configuration Manager (JCM) previously created at BYU. The JCM consists of a custom I/O board paired with a MicroZed development board which includes a Xilinx ZYNQ SoC. This platform is used to implement a flexible configuration management system that can communicate with Xilinx FPGAs at high speeds using the JTAG and SelectMAP interfaces.

The improved system described in this thesis increases the maximum data transfer rate of the JCM's JTAG and SelectMAP interfaces and dramatically decreases the processor utilization of user programs running on the JCM. This is accomplished by incorporating a Direct Memory Access (DMA) engine and interrupts into the system. In addition to faster data rates, these changes and the decrease in processor utilization also allow the JCM to manage up to eight JTAG chains simultaneously with the use of a special I/O card.

Keywords: FPGA configuration, JTAG, DMA, SelectMAP

ACKNOWLEDGMENTS

Dr. Michael Wirthlin has been instrumental in all of the work surrounding this thesis. I would not have been able to complete it without his guidance and insight. I would also like to thank the other members of my graduate committe, Dr. Jeffrey Goeders and Dr. Brent Nelson, for their mentorship throughout my time as a graduate student.

I have also had the support of many other students in the BYU Configurable Computing lab. Ammon Gruwell created the original JTAG Configuration Manager system and helped me get started with the work of this thesis. Luke Newmeyer and Shea Smith are responsible for creating the custom PCBs used in the JCM system. Alex Wilson, Jordan Anderson, and Brittany Wilson have all been of enormous help as well when creating the Linux portions of this work. Finally, Corbin Thurlow has put the new JCM system to use and helped it get to the state it is in now.

Last of all, I need to thank my wife Coralee and our daughter Florence for their encouragement and patience. Coralee has put in even more work at home than I have put into this thesis during the time it has taken to complete it, all while expecting a baby boy. I could not have finished without her support.

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 1265957 through the NSF Center for High-Performance Reconfigurable Computing (CHREC).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be reconfigured to perform a wide variety of functions [1]. Their flexibility and low up-front development costs make them an attractive solution for systems produced on a small scale, including space computing systems such as those found in satellites and other spacecraft. They are also seeing increased use in large-scale applications such as servers and data centers [2].

Many FPGAs use Static Random Access Memory (SRAM) to store data that determines the current configuration of the FPGA. This data must be loaded into the FPGA every time it is powered on. The size of the configuration data of modern FPGAs can reach up to 100 MB, making the speed at which configuration data can be loaded into an FPGA a concern for many designers. This can be even more of a concern in applications where FPGA configuration memory is used during operation for reconfiguration, verification, or other purposes.

One such application is evaluating the reliability of FPGA designs. Reliability is an important consideration in many systems, but especially for those that are deployed in environments with high levels of radiation. Exposure to radiation can potentially corrupt bits within the FPGA configuration memory, changing the behavior of the FPGA [3]. If the memory is corrupted and FPGA behavior changes, the FPGA or the entire system could potentially fail. The effects of radiation and memory corruption on FPGAs is a major area of research conducted in BYU's Configurable Computing Lab.

To better understand and mitigate the effects of FPGA memory corruption, the FPGA configuration memory must be effectively tested and monitored. A common method for accessing FPGA configuration memory is the Joint Test Action Group (JTAG) serial protocol [4]. It is implemented on nearly all Printed Circuit Boards (PCBs) as a means of testing the connections between chips on the board. Because it is present on most PCBs, FPGA manufacturers include ways to interact with their devices using JTAG as well. The JTAG Configuration Manager (JCM) is a system

developed at BYU that communicates with Xilinx FPGAs using JTAG [5]. It can inject faults to simulate memory corruption, scrub memory to fix errors as they occur during testing, and detect memory upsets during radiation tests. These functions can be difficult to manage with commercially available FPGA programmers, but the JCM has been developed from the ground up as a tool to aid in fault injection and radiation testing.

However, some aspects of the design of the JCM significantly limit its performance. In particular, inefficient polling techniques and data transfers limit the rate at which configuration data can be transferred. The work presented in this thesis addresses these issues by adding Direct Memory Access (DMA) and interrupt capability to the existing JCM system. In addition, the SelectMAP interface, a parallel interface for accessing the configuration memory of Xilinx FPGAs, has also been implemented in the JCM.

After these changes, the data rate when transferring data over JTAG using the JCM approaches the maximum possible data rate for a given JTAG clock frequency. The SelectMAP interface also provides data rates of up to 950 Mbps, which is nineteen times the fastest possible data rate using JTAG. The processor utilization during data transfers also decreases dramatically, freeing up processor resources to be used by the Linux kernel or other user processes. This decrease in utilization also allows the JCM to manage up to eight JTAG devices simultaneously using a custom I/O board.

This thesis begins with a chapter describing the details of FPGA configuration and motivating the need for high-speed configuration management. This chapter describes how FPGA configuration memory is stored as well as applications that require access to it. Chapter 3 covers the background of the JCM, including the SelectMAP module that was developed before the introduction of DMA and interrupts in the new system. The performance of the JTAG and SelectMAP interfaces of the JCM are then analyzed and solutions are proposed to increase performance. Chapter 4 gives some background about DMA and then describes the new DMA firmware system that is implemented in the Programmable Logic (PL) portion of the JCM. Chapter 5 describes the changes that were made in the JCM software to support the new features added to the firmware. Chapter 6 presents performance data obtained using the new JCM system and compares it to performance data from the old system. Finally, Chapter 7 concludes this thesis and describes possibilities for future work.

CHAPTER 2. XILINX FPGA CONFIGURATION

FPGAs are powerful and flexible tools for implementing digital logic. They can be reconfigured to perform a wide variety of tasks during the design process and, if necessary, even years after a system is deployed. This chapter describes the details of Xilinx FPGA configuration that must be understood in order to fully appreciate the contributions of this thesis. In particular, the structure of Xilinx FPGAs is described, including how configuration information is stored. Different interfaces for accessing the FPGA configuration data are then presented. Finally, the ways that this information is used during FPGA operation and testing are then described.

2.1 FPGA Structure and Configuration

The heart of an FPGA is an array of programmable logic blocks with configurable signal routing between them. Each logic block contains Look-Up Tables (LUTs), registers, and other basic structures such as multiplexers and adders. Used in combination, these basic elements can implement arbitrary digital logic functions [1]. In addition to programmable logic blocks, modern FPGAs also often include specialized modules such as digital signal processors or multi-gigabit transceivers that can be used to speed up some operations or handle data movement [6]. Such modules are often referred to as hard blocks, as opposed to the soft programmable logic blocks.

Table 2.1: Configuration File Sizes of Xilinx FPGAs

FPGA Part	Configuration File Size
Virtex-4 LX200 [7]	6.12 MB
Virtex-5 TX240T [8]	7.84 MB
Virtex-6 475T [9]	18.68 MB
Virtex-7 1140T [10]	45.91 MB
Virtex UltraScale 440 [11]	122.99 MB
Virtex UltraScale+ 13P [11]	108.07 MB

All of these components must be configured, including routing of inputs and outputs, to correctly implement the user-specified circuit on the FPGA. The data that specifies the current configuration of an FPGA is stored in Static Random Access Memory (SRAM) cells collectively known as configuration memory (CRAM). This data includes programmable logic block configuration, signal routing, I/O buffer configuration, and initial flip flop values among other things. Because SRAM is volatile memory, all of this data must be loaded into CRAM every time the FPGA is powered on. As FPGAs have gotten larger and more complex over the years, the amount of CRAM necessary to contain all of the configuration data has increased dramatically. Table 2.1 shows the size of the configuration file for the largest FPGA part of each generation of Xilinx FPGAs starting with the Virtex-4 series which released in 2005. With configuration files potentially larger than 100 MB in the newest lines of Xilinx FPGAs, loading all of this data into CRAM can take a significant amount of time.

SRAM cells are also vulnerable to a phenomenon called a Single Event Upset (SEU) [3]. An SEU occurs when high-energy particles pass through a transistor, changing its state from off to on or vice versa. This means that CRAM can change unexpectedly during operation, possibly causing a change in circuit behavior and output. For most applications, SEUs are not a concern due to the very low frequency of SEU occurrences or low reliability requirements of a particular system. However, for high-reliability applications or those deployed in environments where more radiation is present, the effects of SEUs on FPGA behavior are important to consider. In order to understand and mitigate those effects, efficient management of CRAM data is essential.

2.2 Accessing the FPGA Configuration

This section covers the basics of accessing the configuration memory of Xilinx FPGAs. The first subsection describes the FPGA configuration registers used when interacting with the configuration memory. The second subsection explains the commands that are sent to the FPGA configuration module via the interfaces described in Section 2.3 when reading or writing these registers. All of the information in this section can be found in [10].

2.2.1 Configuration Registers

The configuration memory of Xilinx FPGAs is modified exclusively through reads and writes to a set of registers. These registers serve various functions, such as option registers that can be written to enable or disable different features of the FPGA or status registers that contain information about the internal state of the device.

This section will only deal with a few of the most commonly used registers: Frame Data Register, Input register (FDRI); Frame Data Register, Output register (FDRO); Command register (CMD); and Frame Address Register (FAR). FDRI and FDRO function as input and output ports for the CRAM. All CRAM data written to or read from the FPGA must pass through these registers. The FAR is written with the address in CRAM that should be accessed through FDRI and FDRO. Finally, CMD can be written with commands that instruct the configuration control logic to perform various operations.

2.2.2 Configuration Command Sequences

All access to FPGA configuration memory in Xilinx devices is handled by a single module in the FPGA. This module receives and executes commands that are received through the various configuration interfaces that are described in Section 2.3. Most of these commands are given in the form of instructions that specify read or write operations on configuration registers. The details of the structure of these read and write instructions, referred to as packets, can be found in [10].

An example of how these commands are used to read from CRAM is shown in Table 2.2. The first command sent is a sync word that must be received by the configuration module before it can accept and process any further commands. After this, a NOOP command is sent before sending a command to write the CMD register. The data to be written to CMD comes immediately after the write command. In this case, it is a command signaling to the configuration module that we want to read CRAM. This is followed by a few more NOOPs before the command to write the FAR and the address to be written to the FAR are sent. Finally, two commands are sent that actually initiate the CRAM read: one command to read FDRO, and another to specify how many words are to be read. After a few more NOOPs, the CRAM data is available to be read through the same interface that has been sending commands. The process of actually reading the requested

Table 2.2: Example Command Sequence to Read CRAM in Xilinx FPGAs

textbfConfiguration Command	**Description**
0xAA995566	Sync word
0x20000000	NOOP
0x30008001	Write CMD Register
0x00000004	Read Configuration Command (Written to CMD Register)
0x20000000	NOOP
0x20000000	NOOP
0x30002001	Write FAR
0xXXXXXXXX	Frame Address
0x28006000	Read FDRO
0x50000065	Read Length Packet (101 Words)
0x20000000	NOOP
0x20000000	NOOP
0xXXXXXXXX	Read data returned from CRAM (101 Words)
...	

data depends on the configuration interface currently in use. These interfaces will be described in the following section.

2.3 Configuration Interfaces

Xilinx FPGAs provide several interfaces for configuration memory access. All of these interfaces use the same set of commands to read and write configuration registers as described in the previous section, but the way these commands are sent and received can vary. This section discusses the Joint Test Action Group (JTAG) and SelectMAP interfaces that are used extensively by the system described in this thesis. These are both external interfaces that provide off-chip access to the FPGA configuration module. Internal interfaces that provide on-chip communication with the configuration module, such as the Internal Configuration Access Port (ICAP) and Processor Configuration Access Port (PCAP), will not be described here because we are only interested in external access to configuration memory. When discussing these interfaces throughout this section, the FPGA is referred to as the slave and the device sending commands to the slave's configuration module through one of these interfaces is referred to as the master.

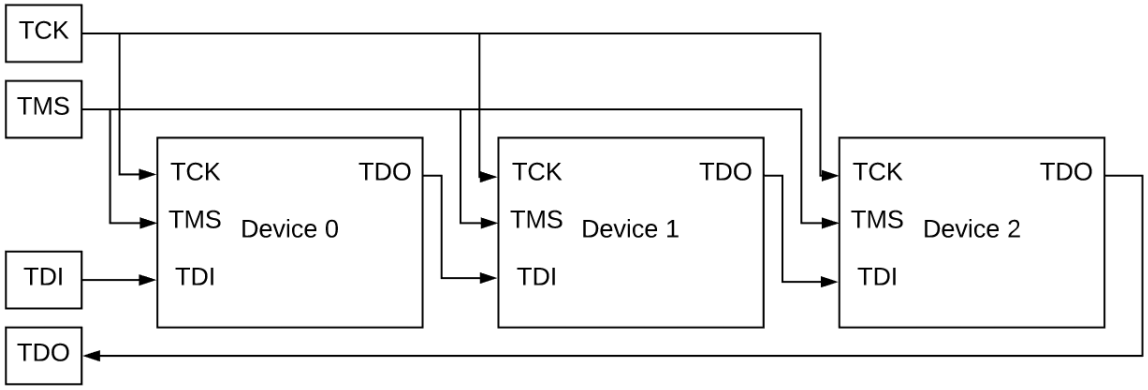

Figure 2.1: Devices connected on a JTAG chain

2.3.1 JTAG

The most common method for accessing FPGA configuration memory is the JTAG interface [4]. JTAG is a serial interface that was originally developed to test connections between chips on a Printed Circuit Board (PCB) to verify that there were no errors during the manufacturing process. It has also been adapted as an external means of communicating with processors and FPGAs for programming and debugging purposes. Practically all modern FPGAs include a JTAG interface that provides easy access for configuration management.

The JTAG interface consists of 4 signals: Test Clock (TCK), Test Mode Select (TMS), Test Data In (TDI), and Test Data Out (TDO). An external device, referred to as the master, can connect to these signals on a PCB and communicate with all of the slave devices on the board. The JTAG master provides the TCK signal during data transfers and uses TMS to change the state of slave devices before transferring data. The TDI and TDO signals transfer data from master to slave and slave to master respectively. These signals are connected to devices on a PCB as shown in Figure 2.1. Devices connected in this manner are referred to as a JTAG chain. Theoretically, there is no limit to the number of devices that could be chained together in this manner. However, we have observed that some JTAG chain configurations limit the maximum TCK frequency due to long signal paths or noisy connections.

In order to communicate with the configuration module of a Xilinx FPGA, the JTAG controller on the slave device must first be put into the proper state. This is done as the JTAG master sends TMS, TCK, and TDI signals to load the instruction register of the JTAG interface with the

correct value. When interacting with the configuration module, the instruction register must be loaded with the CFG_IN instruction when writing data to the slave or the CFG_OUT instruction to read data from the slave as defined in [10]. After the proper instruction is loaded, the master can send or receive data as needed.

A more detailed description of JTAG signals and operation can be found in [12], but the most important fact about JTAG in the context of this work is that all data passing between master and slave is transferred 1 bit at a time through the TDI and TDO signals. This serial transmission, combined with the relatively slow maximum clock speeds and control overhead, means that data transfers using JTAG can be slow. However, the low implementation cost and universal adoption make it an attractive solution for configuration management.

2.3.2 SelectMAP

SelectMAP is an external interface for configuration access available on Xilinx FPGAs [10]. This interface can be used to perform the same configuration management functions as JTAG, but with additional bandwidth by providing more data pins. By using more pins, configuration data can be transferred much more quickly than a serial interface such as JTAG. However, these additional data pins come at the cost of user I/O that could potentially be used for something else. Also, there are many JTAG-specific operations not directly related to configuration management, such as testing connections between parts on a PCB, that are not available using SelectMAP. It can only be used to communicate with the configuration module of Xilinx FPGAs. Because of these limitations, the SelectMAP interface is not as widely used as JTAG and is mostly reserved for applications that require frequent, high-speed configuration management.

The SelectMAP interface consists of an 8-, 16-, or 32-bit bidirectional data bus as well as a Configuration Clock signal (CCLK), a Chip Select signal (CSI), and a Read Write select signal (RDWR). Mode select pins are also used to determine whether the FPGA is a SelectMAP master, meaning it generates control signals, or slave, which only receives control signals. CCLK, CSI, and RDWR are outputs in master mode and inputs in slave mode. The rest of this section will assume that the FPGA is set to slave mode.

The CCLK input must be provided by a device that is acting as master. This device will control the clock and send commands to the slave device while the slave accepts data or responds

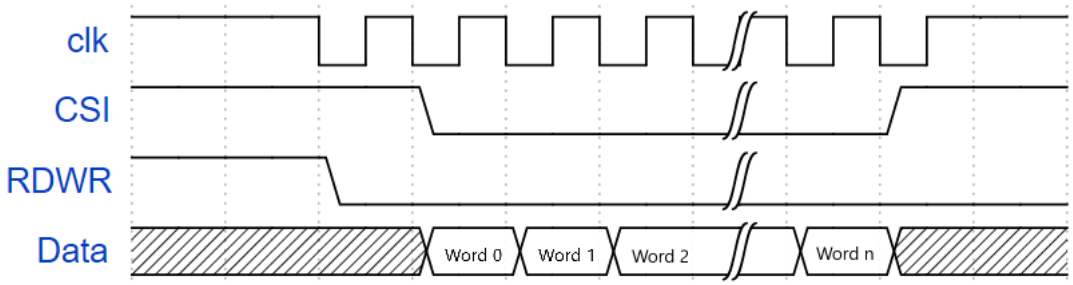

Figure 2.2: SelectMap write procedure (32-bit mode)

accordingly. The waveform in Figure 2.2 shows an example of a write operation using SelectMAP in 32-bit mode. The master holds CSI and RDWR high until one cycle before the transfer begins. RDWR is lowered first to indicate that a write is about to take place and its value is latched by the slave on the rising clock edge. The master then outputs the first word of the transfer on the data pins and lowers CSI on the falling clock edge. Data from the bus is loaded by the slave on the rising clock edge as CSI and RDWR are held low. New data words are output on falling clock edges until the transfer is complete. At this point, the master pulls CSI high after the final rising clock edge and pulses the clock once more to allow the slave to latch the new CSI value[1]. A SelectMAP read is almost identical to the write sequence except that the RDWR signal is held high to designate a read and the data bus changes direction so data is transmitted from slave to master.

The ordering of the data pins is not intuitive and requires some explanation. Each individual byte is bit-swapped, meaning that they are put in reverse order. The mapping for each bit in a word to a data pin is shown in Figure 2.3 for 32-, 16-, and 8-bit modes. When using 16- or 8-bit mode, the most significant portion of the word is output first. For example, imagine we want to send the word 0x12345678. In 16-bit mode, the master would send 0x1234 (0x48C2 after bit swapping) on the first clock and 0x5678 on the second clock. In 8-bit mode, the master would send 0x12 on the first clock, 0x34 on the second, and so on. Data is received from the slave in the same order.

The SelectMAP interface is faster than JTAG for configuration management due to its lack of control overhead and parallel data bus. Table 2.3 shows a comparison of the maximum achievable data transfer rates of JTAG and SelectMAP based on maximum clock rate data obtained from [13]. The high potential data rates make SelectMAP an attractive configuration management

[1]This step is important because it prevents the master from accidentally triggering an ABORT sequence. The ABORT sequence is a feature of the SelectMAP interface that exposes internal state information of a slave device on the data pins for a fixed number of clock cycles. It is triggered when RDWR changes value while CSI is asserted.

SelectMAP Data Bus Width	Data Pins																													
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x32	24	25	26	27	28	29	30	31	16	17	18	19	20	21	22	23	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
x16																	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
x8																									0	1	2	3	4	5	6	7

Figure 2.3: Data pin mapping with bit swapping [10]

Table 2.3: JTAG And SelectMAP Data Rates

Interface	Maximum Clock Rate	Maximum Data Rate
JTAG	66 MHz	66 Mbps
SelectMAP (8-bit)	100 MHz	800 Mbps
SelectMAP (16-bit)	100 MHz	1600 Mbps
SelectMAP (32-bit)	100 MHz	3200 Mbps

solution for Xilinx FPGAs. However, in order to reap the full benefits of SelectMAP, data must be supplied to the interface quickly enough to keep up with the higher transfer rates and provide a continuous stream of data. It is also not commonly implemented on development boards due to the number of pins required to include it on a PCB and the absence of the additional test and debugging capability that JTAG offers. This means that it is often necessary to design a custom PCB to use the SelectMAP interface in an FPGA system.

2.4 Configuration Management Examples

Access to configuration memory is essential for SRAM-based FPGAs because they must be programmed at startup before they can perform their assigned functions. However, programming an FPGA is not the only operation that requires configuration memory access. This section discusses examples of configuration management operations that go beyond just programming an FPGA device with configuration data. It also describes several specific applications in which these operations are used. Finally, several existing configuration management solutions are discussed.

2.4.1 Configuration Management Operations

One common configuration management operation is partial reconfiguration. Partial reconfiguration refers to a design technique in which certain regions of an FPGA device are designed to be reprogrammed at runtime to perform different functions without reprogramming the entire device [14]. This can be used to quickly swap out hardware accelerators during operation or make modifications to algorithms. To do this, special commands, along with the new configuration data, must be sent to the configuration module of the FPGA. The faster these commands can be sent, the less downtime a system will have when switching between configurations.

Memory verification and scrubbing are also common runtime configuration management applications. Readback verification involves reading the CRAM contents and checking them against a golden copy to ensure that the design running on the FPGA matches the expected design [10]. Scrubbing is a related operation which involves actually modifying the CRAM contents so that they match the golden copy [15]. This can be done by doing a readback verification and fixing the parts of memory that do not match (readback scrubbing), or simply by writing the golden data to the device at fixed intervals to ensure that any discrepancies are fixed (blind scrubbing).

Fault injection is a common technique that can be used to evaluate the reliability of an FPGA system. It is the process of manually changing bits in CRAM to simulate SEUs [16]. In general, a fault injection algorithm will flip bits in CRAM sequentially or randomly, wait a fixed amount of time for the error to propagate, and then check if the design failed. This can produce meaningful data, but it can be a very slow process. FPGAs contain millions of CRAM cells, so an exhaustive fault injection campaign can take weeks depending on the error propagation delay between injections. When a design fails, the FPGA must also be reprogrammed. Because this process is repeated so many times, even a relatively small reduction of the time it takes to inject faults and program the device can result in drastically reduced overall test times.

2.4.2 Applications

Aside from fault injection, the primary method for evaluating the reliability of FPGA designs when exposed to radiation is by actually running experiments in radiation testing facilities. This involves placing FPGAs in the path of a radiation beam and observing the effects [17]. Re-

searchers at BYU have conducted many such tests with various collaborators [18] [19] [20]. It is possible to evaluate reliability metrics such as the Mean Time To Failure (MTTF) without using advanced configuration management at these tests by checking the actual outputs of the FPGA design against the expected outputs to determine if a failure has occurred. However, this does not provide any insight into how or why a design actually failed. To answer these questions, a more accurate picture of the state of the device is necessary. To obtain this, a configuration management device can be used to read the CRAM of the FPGA under test and check for SEUs as they occur. These snapshots of the state of the device can help to make sense of the data collected during a test. Scrubbing is also a useful configuration management operation during tests to prevent damage to the device and to see how the introduction of a repair mechanism affects the MTTF and other metrics.

The BYU TURTLE and TORTOISE projects are another example of an application that requires high-performance configuration management. The primary goal of this project is to provide a framework for validating FPGA soft error mitigation techniques such as Triple Modular Redundancy (TMR) using fault injection. The system consists of two FPGAs that are connected in such a way that they can monitor each other's status and detect errors as they occur. The TURTLE does this through a custom FMC board that connects two stock FPGA development boards together, while the TORTOISE is a custom made PCB with two FPGAs on the same board. In both cases, the two FPGAs compare outputs as they operate in lockstep to detect errors that have occurred in one or both of the devices. One of the devices is unmodified and acts as the golden output, while the second FPGA is injected with faults to simulate SEUs. In this manner, the reliability of the design can be characterized with very accurate error detection. However, the process of injecting faults, correcting them, and reprogramming the FPGA when a failure is detected is very time consuming as mentioned previously. In order for this system to produce results in a reasonable amount of time, the configuration manager must be as efficient as possible.

2.4.3 Configuration Management Solutions

As stated previously, the goal of the work presented in this thesis is to provide an external, general purpose configuration manager capable of efficient, high speed data transfers. Many of the solutions described here are either based on internal configuration interfaces or are application-

specific. However, the techniques employed in them are worth noting before moving on to our implementation.

There are several configuration management solutions that have been developed specifically for partial reconfiguration. All of these use internal configuration interfaces such as the Internal Configuration Access Port (ICAP) and Processor Configuration Access Port (PCAP). Both of these interfaces operate with a 32 bit data bus, similar to SelectMAP in 32-bit mode. The Fast Reconfiguration Manager (FaRM) tool described in [21] uses DMA to feed data to a FIFO which then transfers the data to the configuration module using the ICAP interface. With this architecture, they are able to achieve the maximum possible ICAP data rate of 6400 Mbps when writing to the configuration module by overclocking the ICAP module at 200 MHz. The system described in [22] is similar to FaRM but uses the PCAP interface with DMA instead of the ICAP. They are able to achieve a data rate of 3056 Mbps when writing to the configuration module. The speed of these systems is very impressive and the use of DMA to feed high-speed configuration interfaces is promising.

There are also many systems designed specifically for scrubbing and fault injection applications. The system described in [23] successfully implements a scrubber using the ICAP interface. They also mention the possibility of using the SelectMAP interface rather than the ICAP, but they do not explore this option in any of their experiments. A scrubber that uses the PCAP is described in [24]. This system is capable of several different scrubbing modes, including a novel technique which uses the built-in scrubber of Xilinx 7 series devices as well as additional custom logic. Another system described in [15] uses the JTAG interface to implement an external scrubber. The main logic of the scrubber operates on a processor that is connected to an FPGA programmed with a JTAG controller. They are able to achieve a maximum clock rate of 25 MHz.

Each of the systems described above is designed with one specific application in mind and thus lacks the programmability we are looking for in a configuration management solution. Xilinx provides its own programmable JTAG configuration management tools which can be controlled using the TCL scripting language [25]. However, they are still not easily programmable with custom JTAG sequences, relying more on predefined sequences which are sometimes obscured from the user. There also does not appear to be any device on the market or presented in the literature which is capable of providing programmable configuration management using SelectMAP.

The system presented in this thesis is an extension of the JTAG Configuration Manager described in [5]. This was developed at BYU as a programmable configuration management solution that could be used in a variety of situations. The details of this system are covered in the next chapter before discussing the new system presented in this thesis.

2.5 Summary

This chapter has motivated the need for high-speed configuration memory access as well as providing an overview of configuration interfaces available on Xilinx FPGAs. Related work and existing solutions that have influenced this work have also been discussed. The system described in the remainder of this work is designed primarily to utilize the full potential of the JTAG and SelectMAP interfaces for high-speed configuration management in a flexible, programmable environment. The following chapters will describe the details of this system.

CHAPTER 3. JTAG CONFIGURATION MANAGER

The JTAG Configuration Manager (JCM) was developed as a high-speed programmable configuration management solution to meet the needs of our research activities at BYU. It has since become an essential tool in our radiation testing and fault injection work and has also been used at several other companies and universities. The original JCM was largely developed by Ammon Gruwell and is described in [5] and [12].

Figure 3.1 shows the JCM connected to an UltraScale+ MPSoC development board via JTAG. The JCM system consists of two main hardware components. The first is a custom breakout board that provides the JTAG and SelectMAP I/O connections necessary to connect to Xilinx FPGAs. The second is a MicroZed development board which includes a ZYNQ 7010 System on Chip (SoC). This SoC contains a hard processor as well as an FPGA, referred to as the Programmable Logic (PL), which is programmed with hardware IP blocks accessible by the processor. A basic block diagram of the system implemented on the ZYNQ SoC is shown in Figure 3.2. The JTAG and SelectMAP modules in the diagram are IP blocks designed to generate the signals necessary to send and receive data via their respective interfaces. The processor interacts with these modules to handle all communication between the JCM and slave devices.

The ZYNQ processor in the JCM system runs a customized Linux kernel. The Linux environment makes interacting with the JCM easier than a simple bare metal implementation by providing OS functionality like file I/O and Secure Shell (SSH). A large software library that takes care of all interactions with the JCM hardware modules implemented on the PL also makes it easier to modify configuration management functions on the fly during radiation tests or other experiments.

For a deeper look at the details of the JCM JTAG implementation, see Ammon Gruwell's thesis [12]. The focus of this chapter is not to reiterate all of the details of the JCM that have already been published. Instead, this chapter will cover the SelectMAP module that has been

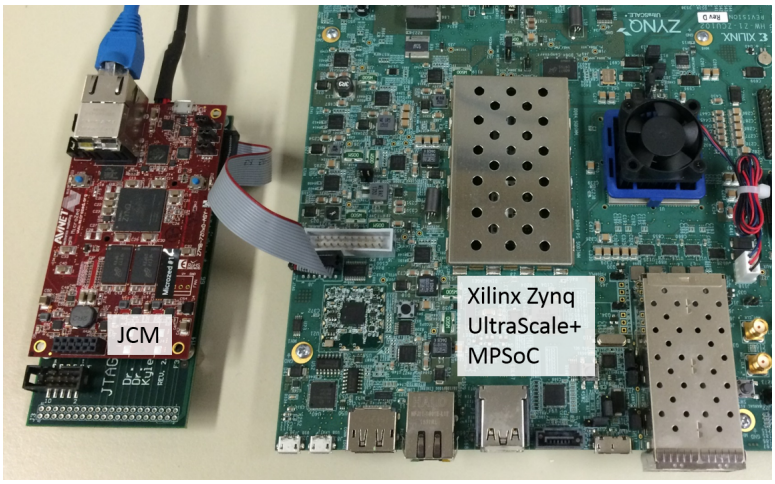

Figure 3.1: JTAG Configuration Manager Connected to UltraScale+ MPSoC

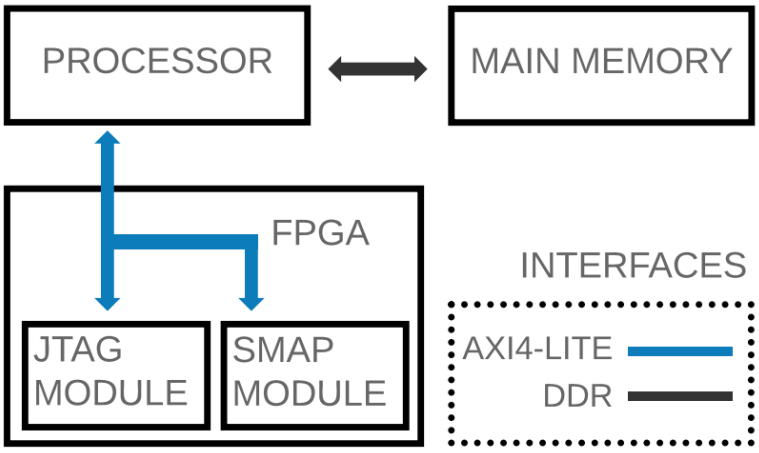

Figure 3.2: JCM System Architecture

added to the JCM since the publication of [12]. The performance of both the JTAG and SelectMAP implementations of the JCM will also be analyzed as well as some specific aspects of the JCM's design that limit its performance. These limitations of the JCM are what prompted the work described in the subsequent chapters of this thesis.

3.1 SelectMAP Hardware Controller

In order to add SelectMAP functionality to the JCM, an HDL module was created as a hardware controller for all of the interface's signals[1]. This module is very similar to the JTAG

[1]Please refer to Chapter 2.3.2 for a detailed look at the signals and basic operation of the SelectMAP interface.

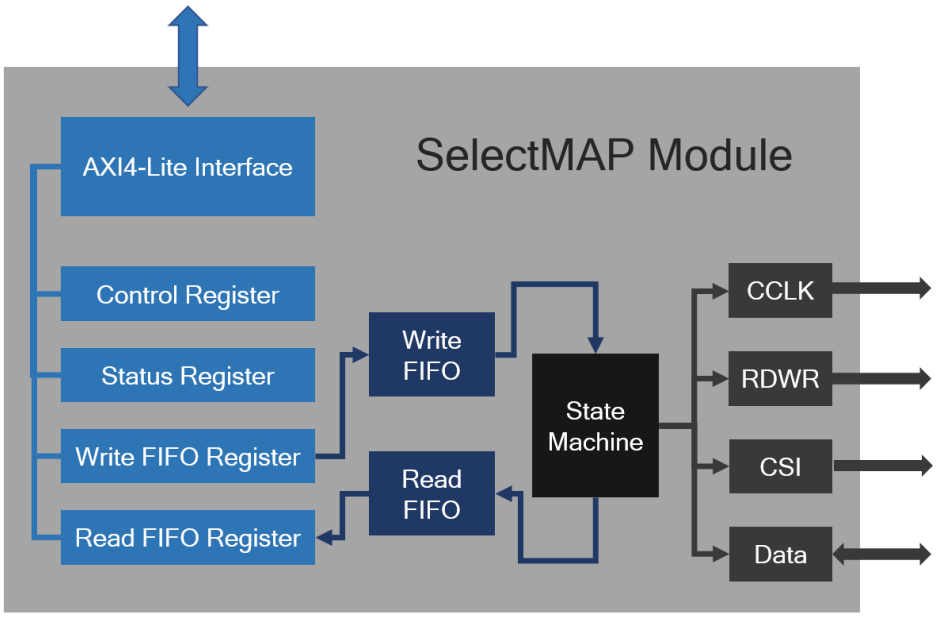

Figure 3.3: SelectMAP Module Block Diagram

hardware controller that is already present in the JCM. A block diagram of the SelectMAP hardware controller is shown in Figure 3.3.

This module is tasked with generating the CCLK, CSI, and RDWR signals as well as providing read and write access to the slave device using an 8-, 16-, or 32-bit data bus. The SelectMAP module is connected to the Zynq processor as a peripheral via the AXI4-Lite Interface. The processor can use this interface to read and write the software-accessible registers shown on the left side of the diagram and control the hardware module. A state machine in the module is responsible for handling all of the SelectMAP I/O signals necessary to send or receive data from the slave. Data that is transferred to and from the slave device is buffered in the read and write FIFOs within the hardware module. The SelectMAP I/O signals are routed to the JCM's custom breakout board. The remainder of this section will describe the details of the registers that are used to interact with the module, the state machine that is responsible for generating and capturing SelectMAP signals, and the software that interacts with the module.

Table 3.1: SelectMAP Module Registers

Register	Description	Offset
Status	Read to obtain information about the state of the hardware module	0x0
Read FIFO	Read to retrieve data that has been received from the slave via SelectMAP	0x4
Write FIFO	Written with data to be sent to the slave via SelectMAP	0x18
Control	Written to specify various options and commands (see Table 3.2)	0x1C

Table 3.2: SelectMAP Module Control Register

Name	Bit Index	Description
Transfer Count	[24:0]	This range specifies the number of 32-bit words to be transferred during this transaction.
Skip Read Wait	25	When set, this bit indicates that the three cycle waiting period before data returning from the slave is valid should be skipped. This is set when the length of a read transfer is larger than the FIFO depth and must be split into multiple transactions.
Config Hold	26	When set, the controller will keep CSI asserted for several clock cycles after the last word of data has been transferred. This is only necessary when programming the FPGA [10].
Hold CS	27	When set, the controller will keep CSI asserted until the next transaction has begun. This is set when the length of a read or write transfer is larger than the FIFO depth and must be split into multiple transactions.
Initiate Transfer	28	This bit indicates that a transfer should be initiated.
Bit Width 16	29	When set, the controller will use a 16-bit data bus.
Bit Width 32	30	When set, the controller will use a 32-bit data bus. Note: If this is set and Bit Width 16 is set, the controller will operate in 32-bit mode. If neither bit is set, the controller will operate in 8-bit mode.
Read/Write	31	This bit specifies whether the current transfer is a read or a write. 0: Read Transfer 1: Write Transfer

3.1.1 Registers

The registers of the SelectMAP module are shown in Table 3.1. There are two software-accessible registers that connect directly to the read and write FIFOs within the SelectMAP module. One of these registers is written with data to be sent to the slave device (Write FIFO), and the other can be read to retrieve data that has been received from the slave (Read FIFO). Both of these FIFOs have a depth of 1024 32-bit words.

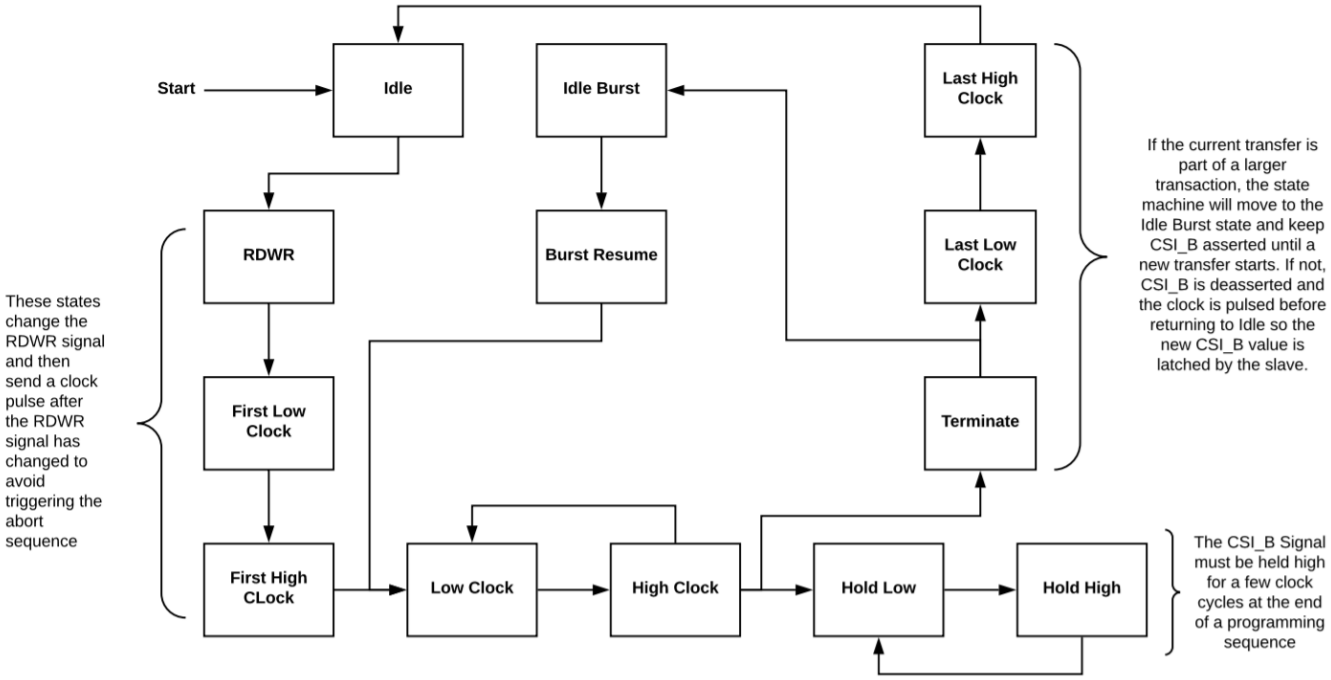

Figure 3.4: SelectMAP State Diagram

The control register in the SelectMAP module is written every time a transfer is initiated. The bits of this control register and their associated functions are shown in Table 3.2. The way that these bits affect the operation of the state machine will be described further in Section 3.1.2.

Finally, a read-only status register provides information about the state of the SelectMAP module. The most important bit, at index 0, indicates whether the SelectMAP controller is currently busy. The remaining four bits are tied to various FIFO flags for debugging purposes, but are unused during normal operation.

3.1.2 State Machine

This section will describe the hardware state machine that is responsible for properly generating and capturing the signals of the SelectMAP interface. A basic diagram of the SelectMAP module's state machine is shown in Figure 3.4. The timing diagram of a SelectMAP write operation shown previously in Figure 2.2 and the read operation shown in Figure 3.5 will also be helpful when trying to understand the operation of the state machine.

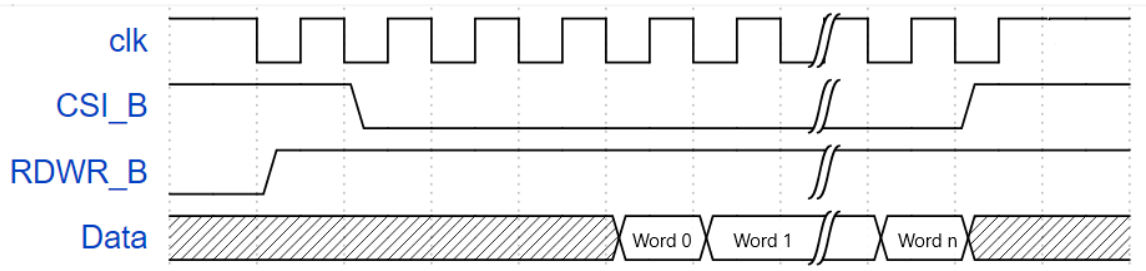

Figure 3.5: SelectMAP Read Operation

The state machine starts in the idle state and remains there until the start bit in the control register is set. In this state, the hardware controller does not interact with the SelectMAP signals. Once the control register has been written with the start command and other options, the state machine moves through three states: RDWR, First Low Clock, and First High Clock. The purpose of these states is to change the RDWR signal to prepare for the transfer without triggering an ABORT sequence. The RDWR signal is changed in the RDWR state to indicate to the slave whether the transfer is a read or write. The clock is then held low in the First Low Clock state and then held high in the First High Clock state so the slave device can latch in the new RDWR signal before CSI is asserted.

At this point, the state machine will alternate between the Low Clock and High Clock states. As the names suggest, the CCLK input to the SelectMAP interface is driven low in Low Clock, and high in the High Clock state. This means that the CCLK signal provided to the slave device operates at half the frequency of the clock input to the SelectMAP module on the JCM[2].

During a write transfer, the next data to be transferred to the slave device is driven onto the data pins during the Low Clock state. During the High Clock state, the data to be written to the slave is still driven on the data pins. This data is held in a FIFO of 32-bit words, but the SelectMAP controller must be able to operate in 8-, 16-, and 32-bit mode. This means that it may take up to four clock cycles to transfer all 32 bits of data retrieved from the FIFO. If the next word of write data from the FIFO is required for the next clock cycle (meaning that the current word has finished transferring), the read enable signal of the FIFO is also asserted during this state.

[2]It is possible to use the SelectMAP interface with a free-running clock rather than a controlled clock, but this has not yet been explored.

In the case of a read transfer, no action is taken during the Low Clock state aside from lowering the CCLK signal. During the High Clock state, the values of the data pins are sampled on the rising clock edge and shifted into a 32-bit register. Once this register is full (every cycle in 32-bit mode, every four cycles in 8-bit mode), its value is stored in the Read FIFO. It is also important to note that the data pins will not be sampled until three CCLK cycles after the CSI signal has been asserted. This is in accordance with the specifications given in [10].

This process repeats until the transfer is complete. This is determined by a counter that is initialized to the transfer count value specified in the control register and then decremented every time a full 32-bit word has been transferred. Once it reaches zero, the transfer is complete. At this point, the state machine diverges depending on whether or not the Config Hold bit of the control register is set. If it is set, the state changes to Hold Low. It will then alternate between Hold Low and Hold High four times while keeping CSI asserted before moving to the Terminate state. If the Config Hold bit is not set, the state machine will move straight from High Clock to Terminate.

In the Terminate state, the Hold CS bit in the control register is checked. If it is high, this means that the current transfer is part of a larger transfer so CSI should be held high until the next transfer begins. The state machine moves to Idle Burst where it keeps CSI asserted and waits for another transfer to begin. If Hold CS is low, the state machine deasserts CSI and moves to the Last Low Clock state. This state, along with Last High Clock, provide one last CCLK pulse before returning to Idle so the slave can latch the deasserted CSI value.

3.1.3 Software

The object-oriented structure of the JCM software is organized so that there are several layers of abstraction between high-level functions, such as reading a particular configuration register or configuring the FPGA with a bitstream, and the direct interactions with hardware modules in the PL. These layers are shown in Figure 3.6. This structure made it relatively simple to replace the JTAG-specific code with SelectMAP code while retaining the high-level operations available in the JCM software.

The top two levels of the JCM software remain unchanged with the introduction of SelectMAP. The application level contains code for performing high-level operations such as scrubbing algorithms, fault injection, and programming an FPGA device. Below that is the device level,

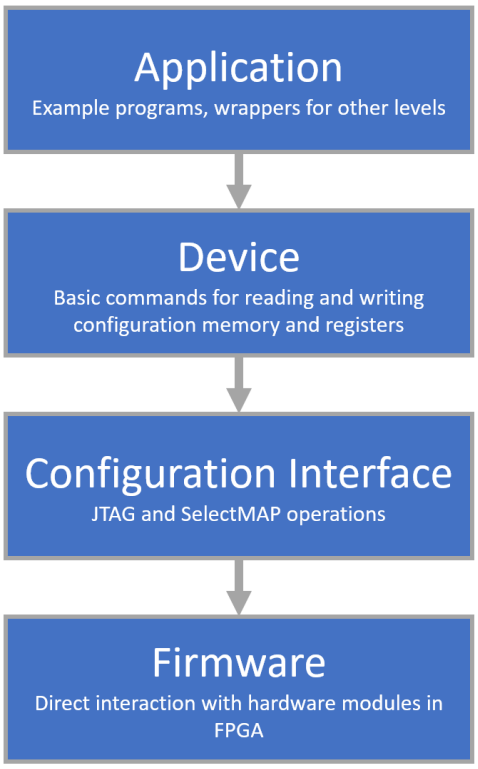

Figure 3.6: JCM Software Layers

which implements low-level configuration operations such as reading and writing configuration memory and registers. This mostly entails assembling sequences of configuration commands that are then passed to the configuration interface level of software so they can actually be sent to the slave device.

The configuration interface level is the highest level in the software that needed modification with the introduction of SelectMAP. To make the SelectMAP and JTAG interfaces interchangeable from the perspective of the higher-level code, an abstract class is used to encapsulate the basic operations of a configuration interface like reads and writes. Separate configuration interface classes for JTAG and SelectMAP which implement those basic functions are then derived from this abstract base class. The high-level software only interacts with objects declared as the abstract base class rather than a JTAG or SelectMAP configuration interface object. In this way, the high-level software functions do not need any knowledge of the specific interface they are interacting with. This means that the same high-level code, such as scrubbing algorithms or fault injection routines, can function properly using JTAG and SelectMAP interchangeably without modification.

Table 3.3: SelectMAP Read ID Code Command Sequence

Configuration Instruction	Description
0xFFFFFFFF	Dummy word
0x000000BB	Bus width auto detect, word 1
0x11220044	Bus width auto detect, word 2
0xFFFFFFFF	Dummy word
0xAA995566	Sync word
0x20000000	NOOP
0x28018001	Read ID code register
0x20000000	NOOP
0xXXXXXXXX	ID code returned from device
0x30008001	Write CMD register
0x0000000D	DESYNC command
0x20000000	NOOP
0x20000000	NOOP

One difference between the JTAG and SelectMAP configuration interface classes is the sequences of configuration commands that are written to the slave FPGA's configuration module. However, the differences are minor so the only change is that the SelectMAP configuration interface class inserts additional configuration commands before and after the sequence of configuration commands specified by the higher-level code.

Table 3.3 shows the sequence of commands necessary to read the ID code register of a Xilinx FPGA. When using JTAG to connect to the FPGA configuration, only the commands in the white cells of the table are needed. When using SelectMAP, the commands in the blue cells must be sent in addition to the commands in the white cells. These additional commands are all taken care of within the SelectMAP configuration interface class so the same sequence of commands can be passed to the JTAG and SelectMAP configuration interface classes without modification. The dummy words are used to flush the packet buffer as specified in [10]. The two bus width auto detect words are sent so the FPGA can determine if the SelectMAP interface is operating in 8-, 16-, or 32-bit mode. At the end of the sequence, the CMD register must be written with a DESYNC command so the SelectMAP interface will release control of the configuration. Finally, two NOOP instructions are sent to flush the packet buffer and let the DESYNC command take effect.

The firmware level of the JCM software is also modified to include a class that interacts with the SelectMAP hardware module directly. The SelectMAP configuration interface class con-

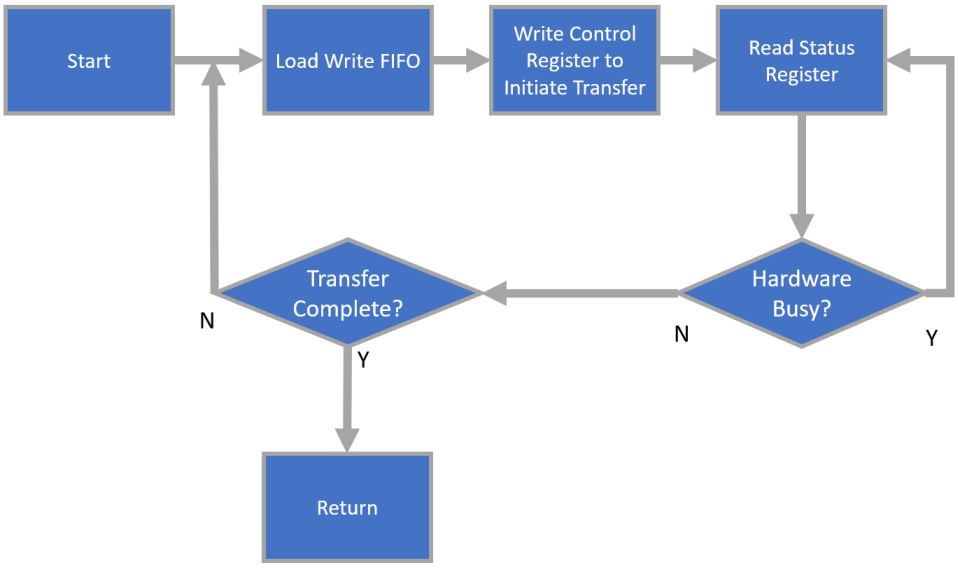

Figure 3.7: JCM Software: SelectMAP Write Flowchart

tains a pointer to an object of this class. The firmware level of the SelectMAP software uses memory-mapped I/O to interact with the registers in the SelectMAP hardware module. This includes interacting with the read and write FIFOs in the module. It also handles longer transfers that exceed the depth of the hardware FIFOs, meaning that multiple transfers are necessary. The flow of the write and read functions is shown in Figures 3.7 and 3.8 respectively.

The write function begins by filling the Write FIFO with data to be sent to the slave via SelectMAP. Then the control register is written to initiate the transfer. After this, the software polls the status register to see if the hardware is still busy with the transfer. Once the hardware is no longer busy with the transfer, the function checks to see if there are still words remaining in the transfer. If so, it fills the FIFO again and repeats the process. If not, it returns.

The read function begins by writing the control register to initiate the read transfer. Then it polls the status register just like in the write function to see if the hardware has finished reading yet. Once the transfer has finished in hardware, the software reads the data from the hardware Read FIFO and saves it in memory. If there are still more words to be transferred, another transfer is initiated and the process is repeated. Once the full transfer is complete, the function returns a pointer to the data that has been retrieved from the Read FIFO.

This concludes the description of the SelectMAP functionality of the JCM. The following sections will examine the performance of both the JTAG and SelectMAP interfaces of the JCM.

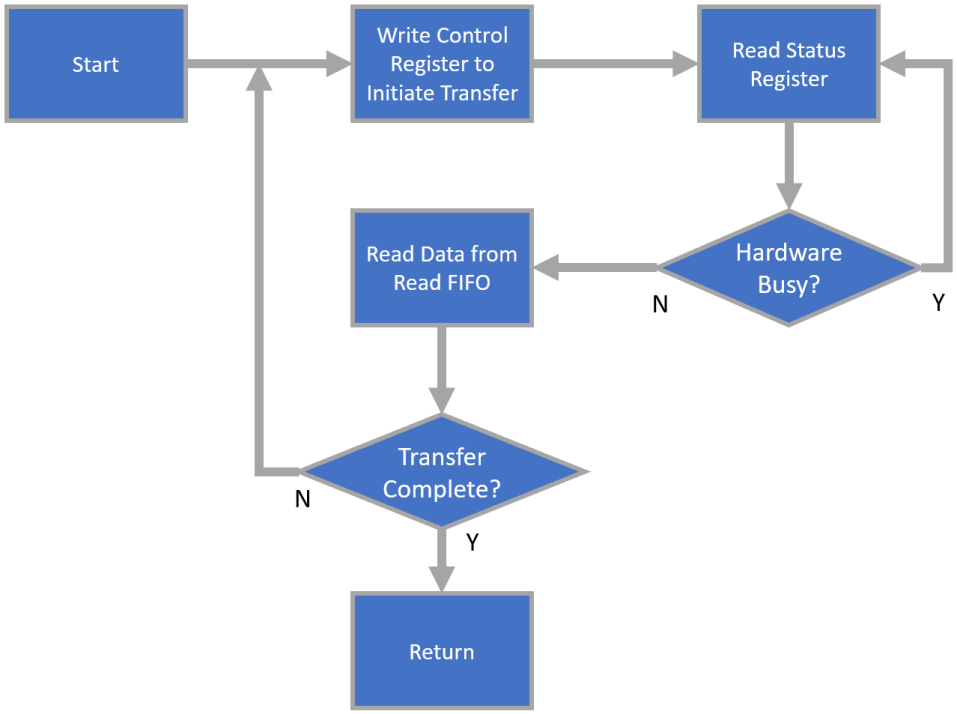

Figure 3.8: JCM Software: SelectMAP Read Flowchart

3.2 Performance

The JCM is already a useful tool that has been deployed at many radiation tests and has also been used to gather valuable fault injection data. However, it has the potential to provide even faster and more efficient configuration management. This section will first define metrics that will be used throughout the rest of the thesis to quantify the performance of the JCM. Then some of the limitations of the JCM's JTAG and SelectMAP implementations will be described along with performance data that highlights the effects of these limitations. Note that all of the performance data presented in this section pertains only to the JCM as it was prior to the work presented in this thesis. Performance data of the enhanced system will be discussed in Chapter 6.

3.2.1 Metrics

There are two main performance metrics that will be used to characterize JCM performance: data rate and processor utilization. The data rate, or the rate at which data is transferred between master and slave devices, will be expressed in Megabits per second (Mbps). This is es-

sential to measure because fast configuration management is the primary goal of the JCM and the work in this thesis. Data rate measurements will be obtained by measuring the amount of time it takes to transfer a large amount of data (800 Mb for JTAG, 4,000 Mb for SelectMAP) between the JCM and a slave device. Using a large amount of data is important to ensure that the overhead of the code used to measure the elapsed time does not significantly influence the measured data rate.

The data rate is linearly related to the frequency of the clock provided to the configuration interface. The JTAG interface on Xilinx devices has a maximum potential frequency of 66 MHz and SelectMAP has a maximum of 100 MHz as shown in Table 2.3. Unfortunately, these frequencies cannot be provided by the JCM firmware when implemented on the ZYNQ 7010 part. The maximum clock speed that can be provided to the hardware controller modules by the ZYNQ 7010 while still meeting timing requirements is 100 MHz. Because the clock supplied to the modules is divided by two before being passed to the slave device, 50 Mhz will be the upper frequency limit in all of the data presented in this thesis. It might be possible to increase this to 100 MHz by using a free running clock in the configuration interfaces rather than generating the clock in the JTAG or SelectMAP state machines, but this has not yet been explored.

The second performance metric is processor utilization. This is defined as the percentage of total execution time during which a process is actively consuming processor resources during a JCM data transfer, as opposed to sleeping or waiting for resources to become available. This percentage is obtained by dividing the CPU time of a process (time during which the process is consuming processor resources) by the total execution time. Using Linux, the total execution time can be obtained using *get_time_of_day()* and the CPU time can be obtained from the *clock()* function. Processor utilization is an important metric to consider because if the utilization is too high and the JCM software hogs the processor, the system as a whole can take a performance hit as important OS processes are forced to wait or preempt user processes at unpredictable times. It also limits the number of user processes that can be active at a time.

3.2.2 Performance

The data rates when using the JCM to write and read data to and from a slave device are shown in Figure 3.9. The percentages above the JCM read and write data rate bars in the graph represent the percentage of the maximum potential data rate that is achieved. Notice that

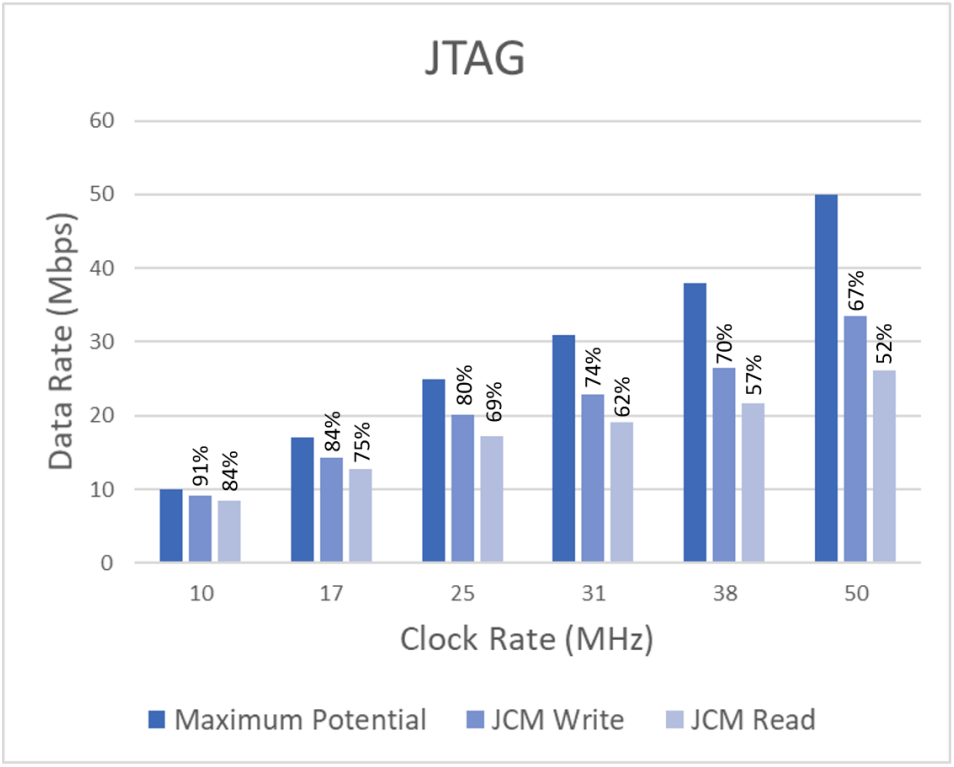

Figure 3.9: JTAG Data Rate Comparison

at low clock speeds, the percentage is close to 100% for write operations. However, as the clock speed increases, the percentage drops dramatically until it is almost have of the maximum potential for read operations and only 67% for write operations. This exposes some serious performance bottlenecks in the design that will be discussed in the next section.

The SelectMAP data rates are shown in Table 3.4. The write and read data rates are practically identical for all clock rates, so only the write data rate is shown here. The same trends that were observed in the JTAG performance are present here as the percentage of the maximum data rate achieved drops dramatically as the clock speed increases. However, in the case of SelectMAP the dropoff starts at slower clock speeds and is even more dramatic. The upper limit of the data rate for any SelectMAP bitwidth and clock speed combination is 99.85 Mbps with a 50 MHz clock and 32-bit data bus.

The measured processor utilization of the JCM software using JTAG or any SelectMAP configuration at any clock speed is greater than 99%. This means that while the JCM is in the middle of a transfer, there are effectively zero processor resources available to any other processes

Table 3.4: SelectMAP Data Rates

	Clock Speed	10 MHz	25 MHz	50 MHz
8-Bit SelectMAP	**Maximum Data Rate**	80 Mbps	200 Mbps	400 Mbps
	Actual Data Rate	45.67 Mbps	69.47 Mbps	84.1 Mbps
	Percentage of Maximum	57.1%	34.74%	21.03%
16-Bit SelectMAP	**Maximum Data Rate**	160 Mbps	400 Mbps	800 Mbps
	Actual Data Rate	63.89 Mbps	84.1 Mbps	94.0 Mbps
	Percentage of Maximum	39.93%	21.03%	11.75%
32-Bit SelectMAP	**Maximum Data Rate**	320 Mbps	800 Mbps	1600 Mbps
	Actual Data Rate	79.85 Mbps	93.97 Mbps	99.85 Mbps
	Percentage of Maximum	24.95%	11.75%	6.24%

in the system. This high utilization of processor resources can starve essential kernel processes, causing them to interrupt the user process for longer periods of time. We have observed this behaviour during radiation tests when a JCM is used to scrub configuration memory continuously for a long time. The time between scrub cycles remains relatively constant, but occasionally the kernel preempts the scrubbing program and the scrub cycle time increases dramatically.

3.2.3 Limitations and Proposed Solutions

Three main performance limitations of the JCM and proposed solutions to these limitations will be described in this subsection. The first limitation applies only to the JTAG interface, but the second and third limitations are common to both the JTAG and SelectMAP interfaces.

The first limitation has to do with the design of the JTAG module's state machine. Currently, the length of a JTAG burst transfer is determined by the number of words that have been written to the Write FIFO in the JTAG module that contains data to be written to the slave. The module will only continue to transfer data between master and slave if that FIFO is not empty. This works well for writing data to the slave, but it slows down burst read operations because the write FIFO must be filled before the read operation can begin. Sometimes this is necessary because the data sent to the slave during a read transfer is used in some way. However, in most cases where large amounts of data are read from a slave device, the values sent to the slave during the transfer are not used at all. This means that the processor must move twice as much data between memory and the hardware modules as is actually necessary for most read operations. The difference

between the data rate for JTAG read and write operations can be seen in Figure 3.9. Notice that at high clock frequencies, the difference in data rates for write operations and read operations is significant. The gap between the read and write data rates widens as the clock speed increases because at low clock speeds, the execution time is dominated by the time it takes to actually send data over JTAG from the hardware module. This makes the effect of the extra transfers between software and hardware less noticeable.

To get rid of this issue, there needs to be a way to specify the length of a read transfer without filling the Write FIFO. This will effectively cut the number of transfers between hardware and software in half.

It is also apparent in Figure 3.9 that the actual data rates for both read and write transfers fall short of the maximum potential data rate at a given clock frequency. This is largely due to to the exclusive use of the AXI4-Lite interface to transfer data between the processor and the hardware modules. AXI4-Lite is a subset of the AXI4 interface that provides low-throughput, memory mapped communication between hardware modules [26]. It is intended mainly for reading and writing registers and it does not include any burst transfer functionality. Every time a single word is transferred using AXI4-Lite, a handshake must take place on an address channel as well as a data channel. When transferring large amounts of data between main memory and hardware modules, such as when programming an FPGA with a bitstream, the overhead of these transfers is significant. That overhead limits the achievable data rate of both the JTAG and SelectMAP modules.

To widen this bottleneck between main memory and the hardware modules, AXI4-Lite needs to be replaced with a more efficient interface with less overhead. The solution implemented in this thesis uses DMA and the AXI4-Stream interface to move data more efficiently. This solution is described in more detail in the coming chapters.

The last problem comes from the way that the software checks to see if the hardware has finished a transfer. The status register of both the JTAG and SelectMAP modules contains a bit that indicates whether the module is currently busy transferring data. In software, this register is polled until that bit is no longer asserted. For short transfers, this is not a significant issue. However, when transferring large amounts of data, the processor can spend more time polling the status register than doing meaningful work. The percentage of execution time that is spent polling during a 100 MB transfer (500 MB for SelectMAP) at various clock speeds is shown in Figure 3.10. For low

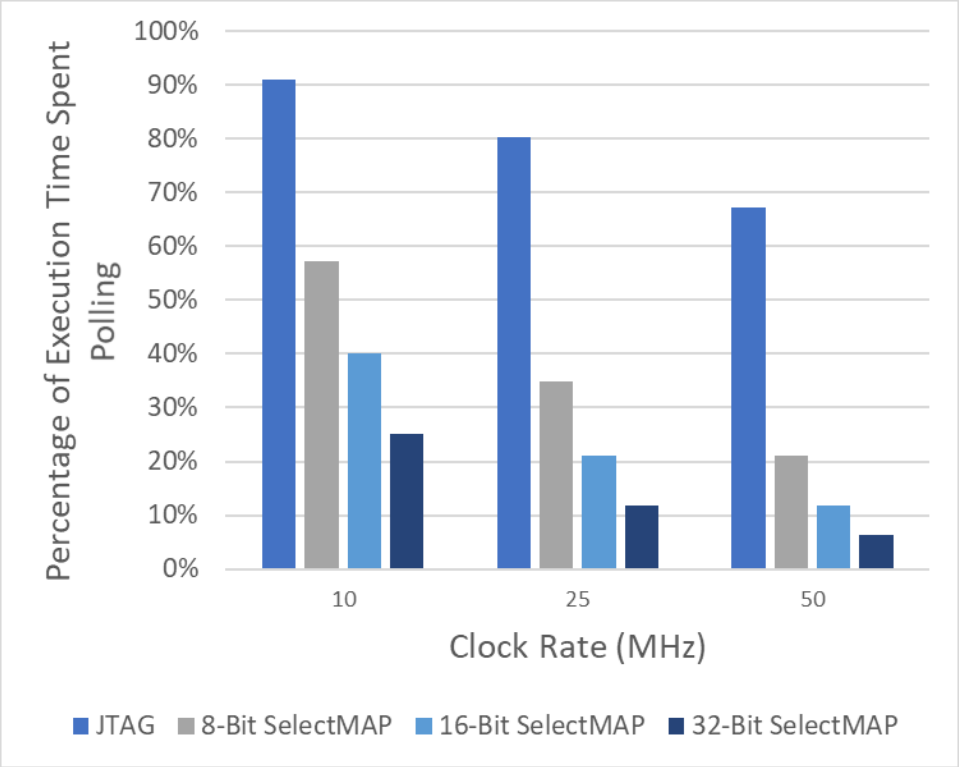

Figure 3.10: Percentage of Execution Time Spent Polling for Different Interfaces and Clock Speeds

clock speeds using JTAG, the percentage of time polling can reach as high as 91%. As the clock speed increases, the percentage goes down because the transfers happen more quickly in hardware so the software does not have to wait as long.

To reduce the processor utilization, the costly polling practices used in the JCM software must be replaced. The solution proposed in the coming chapters uses interrupts so that the user process can sleep while waiting for the hardware to finish transferring data.

3.3 Summary

This chapter has described the state of the JCM prior to the enhancements that are the main contributions of this thesis. It has introduced the SelectMAP hardware module as well as important performance metrics that will be used to compare the base system with the improved system described in the remainder of this thesis. The following chapters will describe this improved system and evaluate its performance.

CHAPTER 4. JCM DMA FIRMWARE

To address the limitations of the JCM described in the previous chapter, a new system including Direct Memory Access (DMA) and interrupts has been developed. The architecture described in this thesis addresses the three limitations of the JCM shown in 3.2.3. This chapter presents the changes to the JCM firmware that solve these problems. In particular, interrupts are used to reduce time spent polling and Direct Memory Access (DMA) capability is added to improve data transfers within the system. A block diagram of the improved system is shown in Figure 4.1. The different pieces of this system and the interfaces between them will be described in this chapter. First, a brief overview of DMA is given. Second, specific details about how it is implemented in the JCM will be presented. Third, the changes to the JCM firmware modules to make them compatible with DMA and efficient interrupt usage are shown. Finally, several implementations of systems with DMA capability and JCM IP are presented.

4.1 DMA Background

Accessing main memory takes a long time compared to most other operations performed by a processor. Not only is it slow, but transferring large amounts of data from place to place can tie up the processor as it executes memory access instructions one by one. DMA is a feature in computing systems which provides access to main memory with minimal processor interaction. This is achieved by adding a hardware module, often called a DMA engine, that can act as a master of the bus which connects various blocks in a computing system. An example of such a system is shown in Figure 4.2. With a DMA engine included on the bus, the processor needs only to specify the details of the transfer to the DMA engine and then it is free to do other work while the DMA engine moves the data between the specified locations. In most cases, the source address, destination address, and number of bytes to transfer are the only parameters required to initiate a DMA transfer. When the DMA transfer is complete, it generates an interrupt to alert the processor.

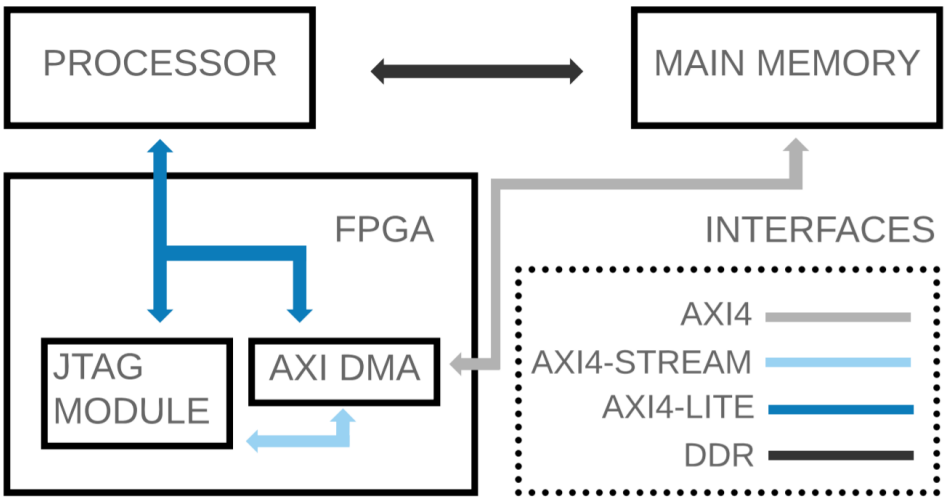

Figure 4.1: JTAG + DMA System Block Diagram

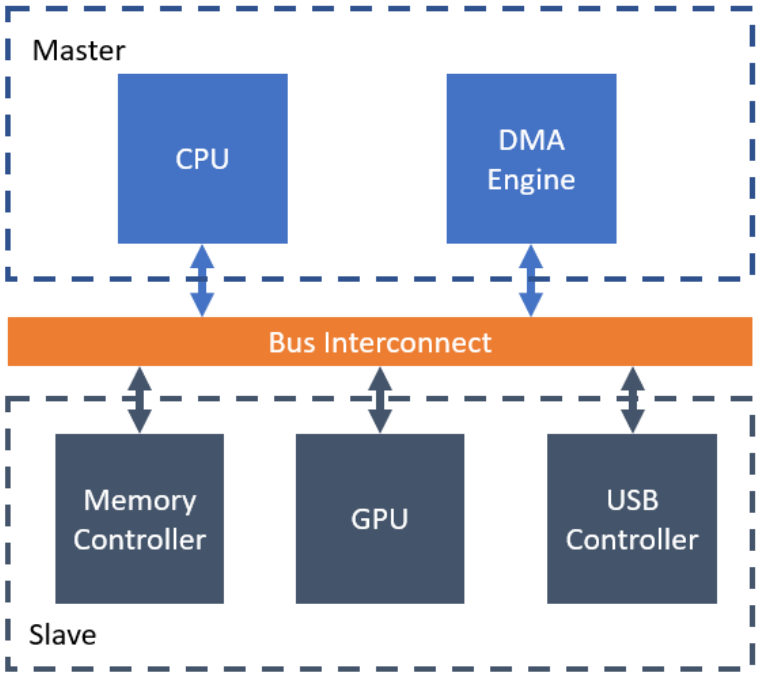

Figure 4.2: Example DMA System

The purpose of the DMA engine in the new JCM system is to move data between the FIFOs of the JTAG and SelectMAP modules and main memory with minimal processor intervention. This will be faster and more efficient than the AXI4-Lite transfers previously used in the JCM to perform the same function. This will ease the memory bottleneck observed in the previous

chapter and should enable faster data rates. It will also reduce processor utilization because the user process can go to sleep while the DMA transfer is in progress.

4.2 AXI DMA

The DMA implementation used in the JCM firmware is the AXI Direct Memory Access IP included in Xilinx's Vivado IP library. It can be configured to provide access to main memory as well as hardware or firmware peripherals within the PL. It also includes two interrupt signals which can be used to avoid polling while DMA transfers are taking place. One fires when a read transfer has completed and the other fires when a write transfer has completed. These interrupt lines can be connected to the interrupt controller of the ZYNQ's hard processor so user processes running on the JCM can sleep while DMA transfers occur.

There are five AXI interfaces of various kinds that are required to use the AXI DMA IP to transfer data between main memory and hardware modules within the FPGA. An AXI4-Lite interface is used to provide the processor with access to control registers within the module. Software running on the processor can then write to these registers to set up DMA transfers. There are two AXI4 interfaces that provide read and write access to main memory via the high performance AXI slave interface on the ZYNQ processor. Finally, there are two AXI4-Stream interfaces that provide read and write access to other IP within the PL. The following section will give a detailed description of AXI4-Stream and why it is efficient for moving data between hardware modules in the PL.

4.2.1 AXI4-Stream

The AXI4-Stream interface is a lightweight data transfer protocol designed for efficient point to point burst transfers [26]. Data is only transferred in one direction and there is no addressing. This means that there is less overhead when initiating a transaction, but it is also less flexible and requires two separate AXI4-Stream interfaces to achieve two-way communication. For the JCM, this is an acceptable tradeoff because data only moves into the JTAG and SelectMAP hardware modules via the Write FIFO and out via the Read FIFO. If reads and writes only ever occur

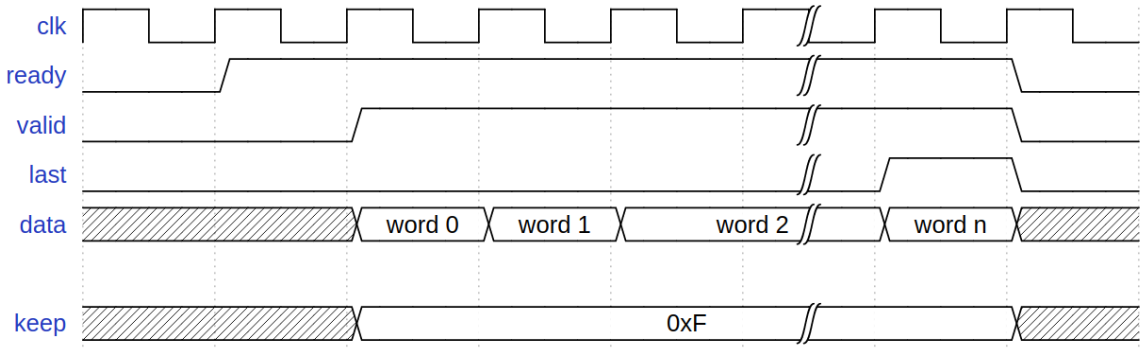

Figure 4.3: AXI-4 Stream Transaction

Table 4.1: AXI DMA Registers

Register	Description	Offset
Source/Destination	The physical address of the source for a DMA write or destination for a DMA read	0x18
Control	Control register which must be written with a start command before a transfer can take place.	0x30
Transfer Length	Length in bytes of the current transfer. Writing this register actually starts the transfer.	0x28

between the same two points, addressing is not necessary and point to point transfers are more efficient.

A typical AXI4-Stream transaction is shown in Figure 4.3. Data is always transferred from a master to a slave that share a common clock associated with their shared AXI4-Stream interfaces. The slave asserts the *ready* signal when it is ready to receive data from the master. The master asserts the *valid* signal once there is data ready to be transferred. At this point, every time *valid* and *ready* are both asserted at a positive clock edge, a single word of data is transferred. At the clock edge which occurs when the last word of data is to be transferred, the master must assert the *last* signal. After this, the master lowers the *valid* signal until there is more data ready to be transferred. Note that the slave is not allowed to wait for the *valid* signal before it asserts the *ready* signal. The *keep* signal determines which bytes of the data bus are valid. In the case of the JCM, all four bytes are valid on every data beat, so *keep* is always driven with 0xF by the master.

4.2.2 Control

The AXI DMA module is controlled by writing to a small set of control registers that are accessible to the processor through memory mapped I/O [27]. These are split into two sets of three control registers: one for the DMA write channel, which handles transfers from main memory to hardware modules, and three for the DMA read channel, which handles transfers in the opposite direction. These registers are described in Table 4.1. The offset column in the table shows the value that must be added to the base address of the read or write channel to calculate the address of the register. The following list shows the necessary steps when beginning a DMA write transaction. All registers mentioned in the list are write channel registers.

1. Write the physical address of data to be written to the source address register
2. Bitwise OR the current control register contents with the start mask value (0x1)
3. Write the transfer length register with the length in bytes of the data to be transferred

The steps to initiate a DMA read transfer are identical except that the read channel registers are written instead of the write channel registers. This also means that the source address register is now a destination address register. This register must be written with the physical address where the data read through DMA is to be stored.

4.3 JTAG and SelectMAP Firmware Changes

This section will describe the changes made to existing JCM IP blocks to support DMA access and interrupts.

4.3.1 JTAG And SelectMAP Modules

The new versions of the JCM JTAG and SelectMAP modules have AXI4-Stream master and slave interfaces that connect to the interfaces on the AXI DMA module. These interfaces replace the memory mapped register interface that was previously used to transfer data to and from the FIFO within these modules. The slave interface on the JCM side which receives data from main memory was very straightforward to implement. The ready signal is always asserted as long

as the internal FIFO is not full. The write clock of the FIFO is connected to the clock associated with the AXI4-Stream interface and the write enable signal of the FIFO is then tied to the logical and of the ready and valid signals. This way, the module is always ready to receive data so long as the FIFO has space available. The *keep* and *last* signals are not used in this application.

The master interface on the JCM modules is more involved due to the generation of the *last* signal. The length of the transfer must be specified to the module in order to assert the last signal on the correct clock cycle, so an additional control register was added. This register must be written with the length of the DMA transfer before the AXI DMA module starts the transaction. A simple state machine within the JCM modules keeps track of how many words have been transferred so far and asserts the last signal along with the final word of data to be transferred.

The behavior of these modules has also changed to more fully take advantage of the higher data transfer rates from memory that DMA provides. The previous JCM version breaks up a single large transfer into a series of transfers that are small enough so that all the required data will fit within the read and write FIFOs in the firmware modules. In the new system, the goal is instead to keep the FIFOs full during a write or empty during a read so that the entire transfer can happen without interruption. The following paragraphs refer to a write transaction where the write FIFO must be kept full, but the same changes also apply to a read transaction and keeping the read FIFO empty.

The main state machine for the JTAG and SelectMAP modules has been modified to reliably support continuous transfers. In the previous version, the JCM modules receive a start command and then send data to the target device until the write FIFO is completely empty. Once the write FIFO is empty, the module returns to an idle state. This means that if the write FIFO is empty at any point in the transaction, the state machine will return to an idle state and the controlling software must write to the control register before it starts transferring data again. The goal for the new version, as stated previously, is to have continuous transfers where the FIFO is not empty until the end of the transaction. However, in the event that the FIFO is completely empty during a write transaction, the state machine should pause until more data is available rather than exit to an idle state.

The new state machine keeps track of the number of words transferred so far to determine whether or not to exit to idle when the write FIFO is empty. If the write FIFO is empty and the

number of words transferred is less than the total length of the transfer (as specified in a control register), the state machine enters a pause state instead of exiting. This allows the transfer to continue without rewriting the control registers.

This change also addresses the inefficiency in burst reads mentioned in Section 3.2.3. When using the previous JCM firmware, the length of read operations is specified by writing the same number of words to the write FIFO as you want to read. For large burst reads, this means that a lot of time is spent sending unnecessary junk data to the write FIFO. By keeping track of transfer length rather than depending solely on the write FIFO's empty flag, read transactions can now take place without writing any data to the write FIFO.

Keeping the write FIFO full also requires that the software knows when to fill the FIFO. This is done by creating a new interrupt signal that is fired when the data in the write FIFO passes below the halfway mark during a write transaction. This same interrupt line is also used when the data in the read FIFO passes the halfway mark during a read. The way this interrupt is handled in software will be detailed in Chapter 5.

4.4 Implementations

This section will describe several systems which take advantage of the DMA engine and other additional features added to the JCM firmware.

4.4.1 JTAG Only

Figure 4.1 from the beginning of the chapter shows the block diagram of a basic system which combines the AXI DMA module with the JCM JTAG module. The figure is simplified slightly so that the interfaces between blocks are easier to follow. In the actual system, the AXI4-Stream interface between the AXI DMA block and the JCM JTAG block is split into two separate interfaces. This is because data can only flow in one direction when using a given AXI4-Stream interface as noted previously in Section 4.3.1. Another difference between the figure and the actual system is that the main memory and AXI DMA modules are not directly connected via AXI4, but are connected through the processor's high performance AXI4 slave port. The AXI DMA connects

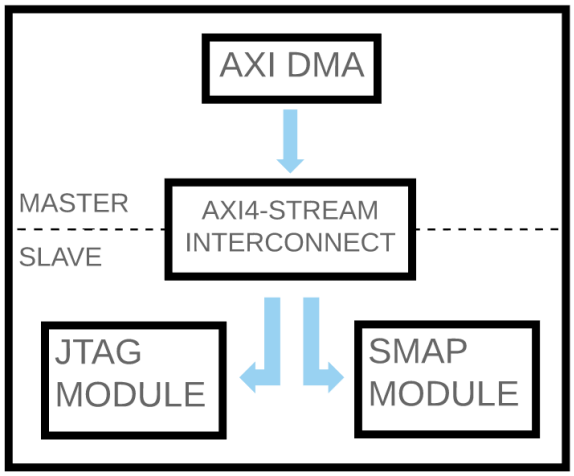

Figure 4.4: AXI4-Stream Interconnect In JTAG/SMAP + DMA System

to a high-performance AXI4 slave interface on the processor, which then provides access to main memory.

There are three interrupts which must be connected to the processor in this system. Two are from the AXI DMA, signaling when a read and write transfer have completed. The final interrupt signal is from the JCM JTAG module, signaling when the internal write FIFO is half empty during a write or the read FIFO is half full during a read. All three of these signals must be routed to the processor and PL interrupts must be enabled when configuring the processing system block in Vivado.

4.4.2 JTAG And SelectMAP

Adding support for a SelectMAP interface in addition to the JTAG interface is not as simple as just inserting the SelectMAP module into the design. Because there are now two modules in the system that must communicate with the AXI DMA module, there must also be a mechanism which can redirect the AXI4-Stream data to the correct destination. The AXI4-Stream Interconnect IP is used for this purpose.

This IP is used to connect one or more AXI4-Stream masters to one or more slaves. Figure 4.4 shows a simplified version of how this system works. The interconnect IP connects to the AXI DMA IP and to both of the modules that will need to communicate with it. Depending on which direction the data needs to move, the AXI DMA will connect on the master or slave side and the

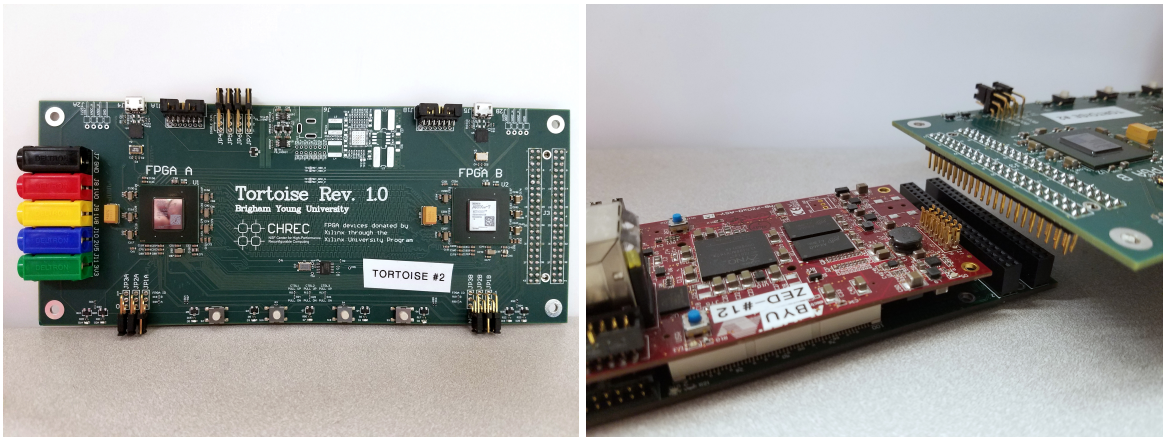

Figure 4.5: TORTOISE Board and SelectMAP Connector

JTAG and SelectMAP modules will connect on the side opposite the AXI DMA. Figure 4.4 shows the case in which the AXI DMA is the master and the JTAG and SelectMAP modules are the slaves, providing read access to memory using DMA. In order for the JCM modules to have both write and read access to memory using DMA, there must be two of these interconnects: one for each direction.

The AXI4-Stream Interconnect IP is controlled by writing software-accessible registers at runtime. There is a register associated with every master interface that is connected to the IP. For each master interface, its associated register must be written with either the index of the slave interface that should be connected to the master interface, or with a constant that disables the interface. After writing any of these registers, a commit command must be written to a control register in the IP before the changes will take effect. The DMA kernel module, which will be explained in Chapter 5, handles all of this to ensure that the AXI DMA is connected to the right device at the right time.

The SelectMAP signals are routed to the accessory connector on the JCM breakout board. From there, several options exist to connect the interface to the SelectMAP interface on the DUT. Individual flying wires are a viable option, but also messy and error-prone. The TORTOISE board, shown in Figure 4.5, was designed to connect directly to the accessory connector on the JCM to provide SelectMAP access. This is an effective solution, but a board must be designed from the ground up in order to take advantage of it. Connecting and disconnecting the two boards must also be done very carefully to avoid bending the pins. With 88 total pins connected via the accessory

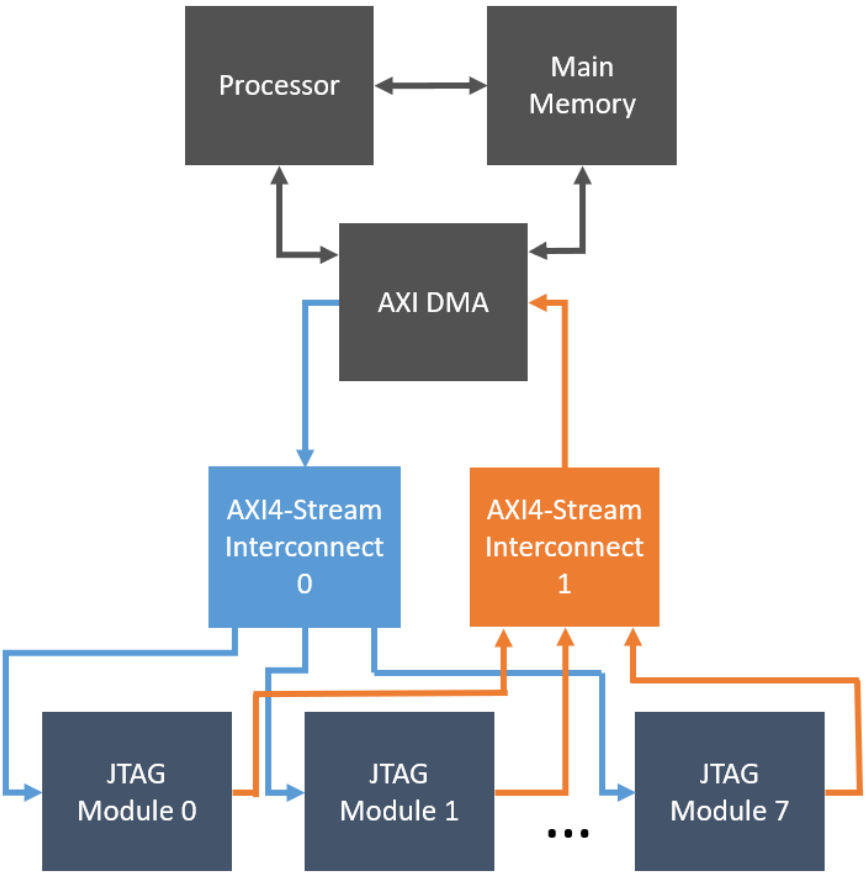

Figure 4.6: Multi-JTAG Simplified Block Diagram

connector, this is actually quite difficult. In the future, an FPGA Mezzanine Card (FMC) connector option will likely replace these initial solutions.

4.4.3 Multi-JTAG

With DMA and interrupts allowing user processes running on the JCM to periodically sleep, processing resources are used more efficiently. This means that managing multiple JTAG chains simultaneously using the same JCM is now much more feasible. The Multi-JTAG version of the JCM firmware was created to make this possible. This version includes eight JCM JTAG modules that are connected to one AXI DMA IP using the AXI4-Stream Interconnect in the same manner that was described in Section 4.4.2. A simplified block diagram of the PL design that implements the Multi-JTAG system is shown in Figure 4.6. The blocks in gray at the top are all

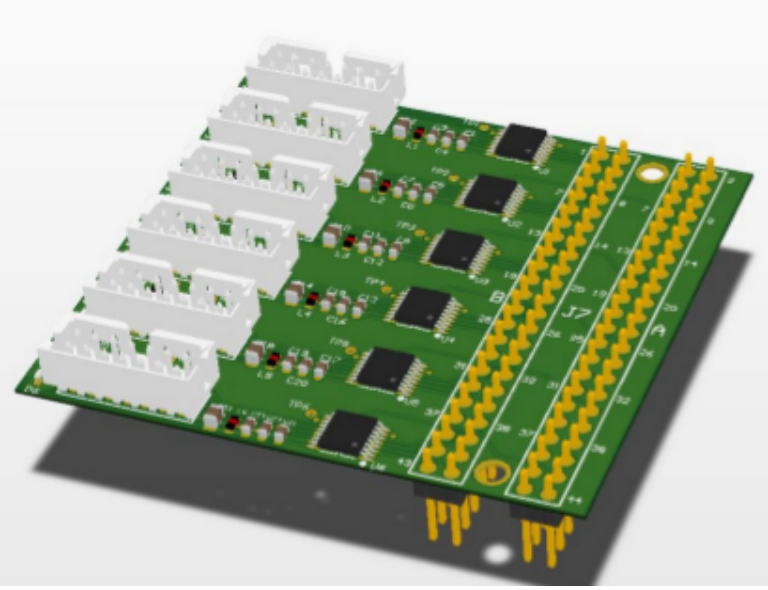

Figure 4.7: Multi-JTAG Expansion Card

connected via high performance AXI4. The two directions of AXI4-Stream interfaces are shown in orange and light blue along with the AXI4-Stream Interconnect blocks that allow the AXI DMA to switch between connections. Finally, the JTAG hardware modules are shown in dark blue along the bottom. The AXI4-Lite connection between the processor and these modules in present in the actual design, but absent from the figure to make it more readable.

The JCM breakout board has also been modified to enable the simultaneous use of eight JTAG connectors. There are two JTAG connectors natively on the JCM breakout board. In addition to these, an expansion card that connects to the expansion slot on the breakout board provides six more connectors. The expansion card is shown in Figure 4.7. By enabling a single JCM to manage eight JTAG chains simultaneously, hardware resources can be used much more effectively. The performance of this system will be shown in Chapter 6.

This chapter has described the changes that have been made to the JCM firmware to add support for DMA and interrupts. While no single piece of the system is particularly complicated on its own, assembling everything into a working system where different blocks interact correctly with each other is a formidable task. The way the software interacts with the firmware is yet another vital piece of this system that will be described in the coming chapter.

CHAPTER 5. JCM DMA SOFTWARE

This chapter describes the changes made to the JCM software to support the new DMA and interrupt features added in the firmware. The majority of these changes take place within the Linux kernel used on the JCM. Specifically, kernel modules are added to the Linux kernel on the JCM to serve as interfaces between software and hardware. These modules are referred to as drivers. This chapter justifies the need for kernel changes by describing the differences between kernel space and user space. Then, the mechanisms used to modify the kernel, namely the device tree and kernel modules, are described. Finally, the specific kernel modules developed for the JCM are discussed in detail.

5.1 Kernel Space

Kernel space refers to the region of memory where the actual Linux kernel code operates and user space refers to the region of memory where user programs operate [28]. However, the region of memory is not the only difference between the two. Kernel space and user space code also executes with different levels of privilege on the processor. These privilege levels, referred to as rings, determine what functionality is available. Although more than two rings exist in most processors, only the outermost (least privileged) and innermost (most privileged) rings are used in Unix systems. User space code operates in the outermost ring and kernel space code operates in the innermost ring. This means that kernel space essentially has no restrictions on operations it can perform and user space is more limited.

It is important to note that the lack of restrictions in kernel space does not necessarily make programming easier. Writing code to run in kernel space can be difficult because a lot of functionality designed for user space, such as C standard library functions, is unavailable in kernel space. Debugging is particularly difficult because the safety net of the kernel is taken away. Errors that occur in kernel space can easily crash the entire system and require a reboot to recover.

Standard debugging tools like GDB also cannot be used in the same way when debugging kernel code. This makes for a very steep learning curve when trying to write kernel code for the first time. With all of these drawbacks, it is generally wise to look for solutions in user space before jumping to kernel space. However, for this system there is ample reason to write kernel code because of the inclusion of DMA and interrupts.

Using a DMA engine requires manipulation of physical memory, which is handled exclusively in kernel space. User space programs are confined to processes that are managed by the kernel to ensure that they do not interfere with each other. Each process has a virtual address space that is mapped to physical memory by memory management code in the kernel. This type of memory management is problematic for a user space program that wants to use DMA because the DMA controller in hardware only uses physical addresses and has no knowledge of the virtual address space used in a process. Also, memory that appears to be contiguous in the virtual address space of a user process could actually be spread out through different parts of physical memory. This is a problem for standard[1] DMA controllers that depend on contiguous chunks of memory when they transfer multiple bytes of data. Due to these obstacles, allocating DMA buffers and handling DMA transfers is best suited for kernel space code.

In addition to the management of physical memory, registering and handling interrupts also occurs within kernel space. The new JCM firmware generates several different interrupts that can be used to utilize processor resources more effectively if handled properly. Before this can happen, the interrupts must be registered with the kernel and associated with an Interrupt Service Routine (ISR) that will execute every time the interrupt occurs.

Considering all of this, writing kernel code to handle the new DMA and interrupt features is essential for the new JCM system. This code is contained in kernel modules, which will be explained in Section 5.3. Before developing the modules, however, the kernel needs to have information about the device that it will be interacting with. This is done through the device tree.

[1]DMA controllers that feature a scatter-gather mode can deal with noncontiguous memory and efficiently transfer data that is spread out through the physical address space, but it makes interacting with the DMA controller more complicated and it is not necessary if there is a sufficient amount of contiguous physical memory available.

```
/ {      // the root node
    an-empty-property;
    a-child-node {
        array-prop = <0x100 32>;
        string-prop = "hello, worlds";
    };
    another-child-node {
        binary-prop = [00102CAFE];
        string-list = "yes", "no", "maybe";
    };
};
```

Figure 5.1: Device Tree Source Example

5.2 Device Tree

To interact properly with hardware devices, the kernel needs to know some fundamental information about them. A device tree is a data structure used to convey this information to the kernel by describing the hardware devices present in an embedded computing system [29]. The kernel loads this data structure at boot time so that it can properly configure the kernel and load device drivers as necessary. Desktop and server environments can avoid using this due to standardized firmware interfaces that can be probed at boot time so the system is aware of attached hardware. However, the device tree is important for embedded systems because the type of hardware available in the system can vary wildly from one system to another and is not as uniform as hardware intended for use on desktops or servers. This is especially true in heterogeneous systems involving CPUs and FPGAs because the FPGA can be reconfigured to contain a wide variety of different hardware modules. The alternative to device trees is to hard code information about the system into the kernel itself, which requires the kernel to be recompiled whenever the hardware changes. Device trees can be changed much more easily, which enables more flexible systems.

A device tree is represented in a readable text format as a device tree source (*.dts*) file. A simple example of a *.dts* file is shown in Figure 5.1. The *.dts* file contains a list of devices present in the system that are organized into nodes. A single root node encompasses all of the other nodes in the entire tree. Common child nodes of this root node include CPUs, memory, and programmable logic. The leaf nodes that have no children represent actual devices present in the system.

Each node in the device tree includes a list of properties which can contain different types of data. These properties can specify memory address spaces for memory-mapped IO, interrupt information, or any other information that the kernel needs to know about the device. Every node also includes a string property with the *compatible* label. This is a unique identifier so the kernel can determine what device is associated with that node.

The number of necessary properties in each node can be large, so creating an entire device tree source file by hand is tedious and error prone. Fortunately, Xilinx's Vivado design suite includes tools for generating device tree source files based on a hardware design. This can be done by exporting the Vivado block design to the Software Development Kit (SDK) and creating a device tree project based on the exported files. Before the device tree can be used by the kernel, it must be compiled into a device tree blob (*.dtb*) file using a device tree compiler. This compiler is also available through Xilinx for use with Xilinx SoCs such as the ZYNQ 7010 used in the JCM. A more specific walkthrough of the process of creating a usable device tree using Xilnix tools can be found at [30].

The device tree nodes that have been added to the JCM's device tree are shown in Figure 5.2. The auto-generated information that is not directly used by the kernel modules has been omitted for the sake of space. From top to bottom, these nodes correspond to the AXI DMA IP (including the read and write channels as separate nodes), the two AXI4-Stream Interconnects, the JTAG module, and the SelectMAP module respectively. The additional seven JTAG modules used in the Multi-JTAG implementation of the JCM hardware are also present, but not shown in this figure.

This section has covered the basics of device trees as they are used in the Linux kernel and the specific changes to the device tree in the new JCM system. The remainder of this chapter is dedicated to Linux kernel modules and how they are used in the new system.

5.3 Kernel Modules

A kernel module is a piece of code that can be inserted into or removed from the kernel on demand [28]. This means that code can be added to the kernel without recompiling the entire kernel, making the kernel more flexible. In the case of the JCM, this is desirable because there are several different hardware configurations used on the FPGA as described in the previous chapter.

```
dma@40400000 {
        compatible = "xlnx,axi-dma-1.00.a";
        interrupt-parent = <0x4>;
        interrupts = <0x0 0x1d 0x4 0x0 0x1e 0x4>;
        reg = <0x40400000 0x10000>;

        dma-channel@40400000 {
                compatible = "xlnx,axi-dma-mm2s-channel";
                interrupts = <0x0 0x1d 0x4>;
        };

        dma-channel@40400030 {
                compatible = "xlnx,axi-dma-s2mm-channel";
                interrupts = <0x0 0x1e 0x4>;
        };
};

axis_switch@43c10000 {
        compatible = "xlnx,axis-switch-1.1";
        reg = <0x43c10000 0x10000>;
};

axis_switch@43c20000 {
        compatible = "xlnx,axis-switch-1.1";
        reg = <0x43c20000 0x10000>;
};
jcm_jtag_module_0: jcm_jtag_module@79c00000 {
        compatible = "xlnx,jcm-jtag-module-1.0-0";
        interrupt-parent = <0x4>;
        interrupts = <0x0 0x1f 0x4>;
        reg = <0x79c00000 0x10000>;
};
jcm_smap_module@43c00000 {
        compatible = "xlnx,jcm-smap-module-1.0";
        interrupt-parent = <0x4>;
        interrupts = <0x0 0x20 0x4>;
        reg = <0x43c00000 0x10000>;
};
```

Figure 5.2: JCM Device Tree Nodes

By implementing the software interfaces for the JCM hardware using modules, the set of interfaces used in the kernel can be changed without recompiling the kernel and creating a new disk image.

Kernel modules can serve different purposes, but the modules developed for the new JCM system are known as device drivers. These are modules that are specifically intended to provide a software interface to hardware devices. When a device driver is inserted into the kernel, several

things happen. First, an initialization function in the driver is called which registers the device with the kernel. Then the initialization function calls a probe function defined in the driver that probes the device tree for information about the device associated with the driver. This is where the compatible property in the device tree comes into play. The probe function executes once for each node in the device tree for which the compatible property matches a string in the match table[2] defined in the driver. The probe function can then pull information such as address space and interrupt numbers from the device tree for each device that the driver needs to interact with. After the probe function has run for the last time, the initialization function can return and the driver is now inserted into the kernel.

When a driver is inserted into the kernel, a device file is created in the kernel's file system. Software running in user space can then perform operations on this file using the Input/Output Control (IOCTL) function. In the kernel module, the IOCTL function defines different operations that can be performed on the hardware device and associates each operation with a unique number. This function is called from user space using three parameters: a file descriptor of the associated device file, the number of the IOCTL operation to execute, and a parameter of type long that can be cast to whatever data type is needed to pass necessary information between user space and kernel space.

5.4 JCM DMA Linux Drivers

There are three distinct device drivers that have been developed for the new JCM DMA system: a DMA driver, JTAG driver, and SelectMAP driver. This section will describe each of these drivers in detail, including how user space software interacts with them. The connectivity of these drivers is shown in Figure 5.3 for the hardware system that includes a JTAG module and a SelectMAP module. When the Multi-JTAG hardware is used, the SelectMAP driver is not loaded and seven additional JTAG modules are loaded instead. The user space code interacts with the JTAG and SelectMAP drivers solely through the IOCTL function. These drivers communicate with the JTAG and SelectMAP firmware modules as well as the DMA driver in kernel space. The DMA driver is never accessed from user space directly. When any of these drivers are loaded,

[2]The match table is an array of C structures which each contain a string member named "compatible." This member is what is compared with the compatible properties of device tree nodes.

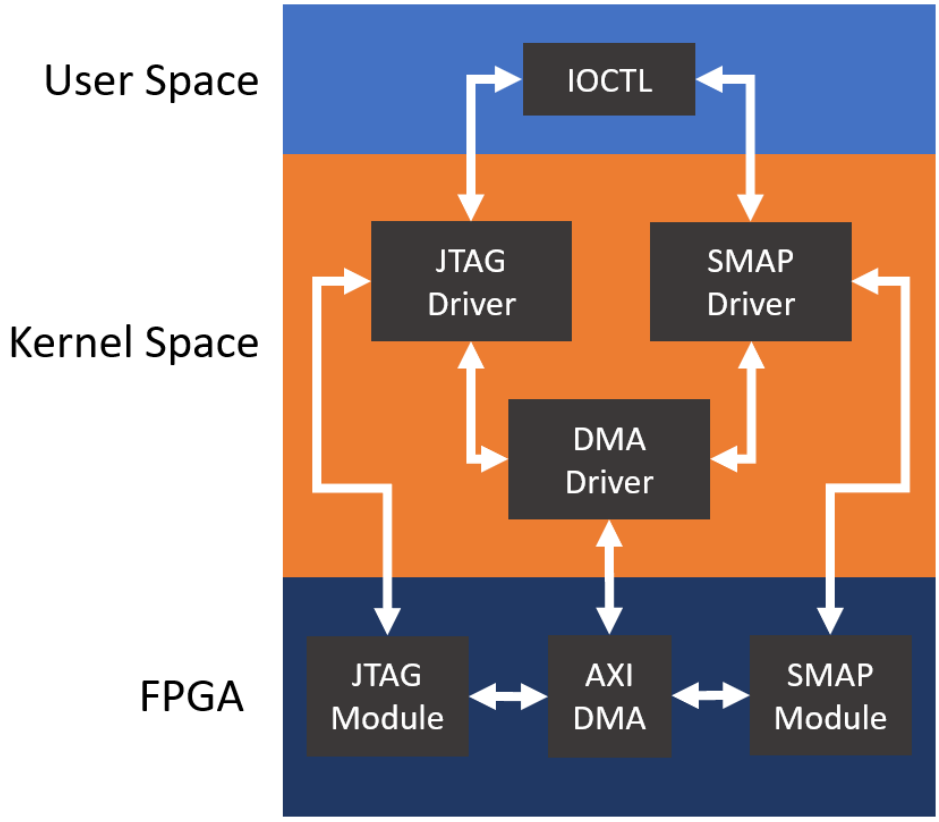

Figure 5.3: User Space, Kernel Space, and Hardware Connections

they probe the device tree to obtain base addresses for the hardware modules involved in the driver as well as any necessary interrupt numbers. The interrupts are then immediately registered with associated ISRs. The specifics of these drivers are covered in the following subsections.

5.4.1 DMA Driver

The DMA driver is responsible for managing all DMA transfers in the system. To do this, it needs to get the base addresses of the AXI DMA IP and the two AXI4-Stream Interconnect modules from the device tree. It also needs to get the interrupt numbers for the read and write channels of the AXI DMA IP. Finally, it allocates a contiguous buffer of one full page (4 KB) of memory to use when transferring data via DMA. After all of the initial setup, the DMA driver exports two functions that are accessible to the other drivers in kernel space: DMA read and DMA write.

Both of these functions are made atomic by using a semaphore. Only one device is allowed to initiate a DMA transfer at a time because every DMA transfer must be preceeded by setting up the AXI4-Stream Interconnects so the AXI DMA IP is connected to the correct firmware module. This connection cannot be changed in the middle of a transfer or the data will be redirected. The AXI DMA IP also only includes one read channel and one write channel, so multiple transfers cannot happen simultaneously.

The DMA read function takes four parameters: a destination address, transfer length, device ID, and DMA control register address for the requesting device. The destination address is used to copy data from the DMA buffer in kernel space to an array in user space so the data read from the slave device can be used there. The transfer length specifies the length of the transfer given in 32-bit words. The device ID is used to determine how to configure the AXI4-Stream Interconnects. Finally, the DMA control register address must be written with the transfer length as described in the previous chapter.

The actual DMA read function is shown in Figure 5.4. First, the semaphore is obtained. Then the AXI4-Stream interconnects are configured according to the device ID. The DMA control register address is then written with the number of words to be transferred. Finally, the AXI DMA control registers are written to initiate the transfer. The process which called the function is then put to sleep until the interrupt indicating that the read transfer is complete has fired. Once the process wakes up, it copies the data that was read from the buffer in kernel space to user space and releases the semaphore.

The DMA write function is very similar and is shown in Figure 5.5. It first obtains the semaphore, then configures the AXI4-Stream interconnects using a device ID passed to the function. It then copies the data to be written to the slave from user space into the buffer in the kernel. The transfer is initiated and then the process goes to sleep until the interrupt signalling the end of the transfer is received. Finally, the process wakes up, releases the semaphore, and returns.

These two functions are never accessed from user space. They are only called from other parts of the kernel, namely the JTAG and SelectMAP drivers that oversee transactions between the JCM and a slave device.

```
int jcm_dma_read(u32* dest_addr, u32 transfer_length, u32 dev_id, u32
    dma_control_addr){
        long ret;
        //Obtain semaphore
        if(down_interruptible(&sem))
                return -ERESTARTSYS;
        connect_axi_stream(dev_id);
        dma_read_ready = 0;      // reset the ready flag

        //Write dma control register in requesting device
        *(volatile u32 *) (dma_control_addr) = transfer_length;
        //Write the destination address (Kernel Array)
        *(volatile u32 *) (dmaBaseAddress + DMA_DEST_ADDRESS_OFFSET) = (u32)
    physAddr;
        //Start the DMA
        *(volatile u32 *) (dmaBaseAddress + DMA_READ_CONTROL_OFFSET) = *(
    volatile u32 *) (dmaBaseAddress + DMA_READ_CONTROL_OFFSET) |
    DMA_START_MASK;
        //Write the Length of the transfer (actually starts transfer)
        *(volatile u32 *) (dmaBaseAddress + DMA_READ_TRANSFER_LENGTH_OFFSET) =
     sizeof(u32) * transfer_length;

        //Wait until transfer is complete
        if(!wait_event_interruptible_timeout(wq, (dma_read_ready!=0), HZ/10))
                printk("DMA read timeout");

        //Transfer data from kernel array to user space
        ret = copy_to_user((u32 *) dest_addr, (u32 *) kernArray, sizeof(u32) *
     transfer_length);
        if(ret < 0){
                printk("Error copying to user space in dma_read\n\r");
        }
        //Release semaphore
        up(&sem);
        return 0;
}
EXPORT_SYMBOL(jcm_dma_read);
```

Figure 5.4: DMA Read Function in DMA Driver

5.4.2 JTAG and SelectMAP Drivers

The JTAG and SelectMAP drivers perform the same operations with very few differences. They are responsible for initiating and managing transfers in the JTAG and SelectMAP firmware modules. They each define the same four IOCTL operations: write, read, register write, and register read.

```
int jcm_dma_write(u32* src_addr, u32 transfer_length, u32 dev_id){
        long ret;
        //Obtain semaphore
        if(down_interruptible(&sem))
                return -ERESTARTSYS;
        connect_axi_stream(dev_id);
        dma_write_ready = 0;     // reset the ready flag

        ret = copy_from_user((u32 *) kernArray, (u32 *) src_addr, sizeof(u32)
    * transfer_length);
        if(ret < 0){
                printk("Error copying from user space in dma_write\n\r");
        }
        //Initiate DMA Transfer
        //Write the Source Address (Kernel Array)
        *(volatile u32 *) (dmaBaseAddress + DMA_SOURCE_ADDRESS_OFFSET) = (u32)
     physAddr;
        //Start the DMA
        *(volatile u32 *) (dmaBaseAddress + DMA_WRITE_CONTROL_OFFSET) = *(
    volatile u32 *) (dmaBaseAddress + DMA_WRITE_CONTROL_OFFSET) |
    DMA_START_MASK;
        //Write the Length of the transfer (actually starts transfer)
        *(volatile u32 *) (dmaBaseAddress + DMA_WRITE_TRANSFER_LENGTH_OFFSET)
    = sizeof(u32) * transfer_length;

        //Wait until transfer is complete
        if(!wait_event_interruptible_timeout(wq, (dma_write_ready!=0), HZ/10))
                printk("DMA write timeout");
        //Release semaphore
        up(&sem);
        return 0;
}
EXPORT_SYMBOL(jcm_dma_write);
```

Figure 5.5: DMA Write Function in DMA Driver

The register read and write functions are exactly as one would expect. They read or write a register in the address space of the hardware module associated with the driver. This is done by making an IOCTL call in user space, passing in the appropriate IOCTL number for the operation, and supplying a pointer to a struct containing an offset and a pointer to a 32-bit unsigned integer as the third parameter. The offset is added to the base address obtained from the device tree to form the address of the register. The pointer is used to obtain data to be written in the case of a register write, or store data read from the register in the case of a register read. These functions

```
while(completed_length < total_transfer){
        // Select how many words to transfer
        current_transfer = ((completed_length + DMA_WRITE_TRANSFER_LENGTH) >
    total_transfer) ? remainder_words : DMA_TRANSFER_LENGTH;

        //Write the next set of words and increment address and counter
        //The while loop is to retry if the write fails
        while(jcm_dma_write(working_address, current_transfer, dev_id));
        completed_length += current_transfer;
        working_address += current_transfer;

        // Done writing. Exit loop
        if(completed_length == total_transfer)
                break;

        //Start the transfer if we haven't already
        if(!started_transfer){
                //Initiate JCM write
                start_jcm(JTAG_WRITE, total_transfer, control_word);
                started_transfer = true;
        }
        // If we have written enough to clear the interrupt, wait for
    interrupt
        if(jtag_interrupt_is_cleared()){
                //Wait until FIFO is empty enough
                wait_for_jcm_interrupt();
        }
}

if(!started_transfer){
        //Initiate JCM write
        start_jcm(JTAG_WRITE, total_transfer, control_word);
        started_transfer = true;
}
//Polling in user space to finish up transfer
break;
```

Figure 5.6: JTAG Write IOCTL Operation

are available so the user space code does not have to deal with the base address of the hardware modules or the specifics of reading and writing registers in the modules.

The write operation is used to write data to the slave device (connected via JTAG or SelectMAP) using the firmware modules described in the previous chapter. In this case, the third IOCTL parameter is a pointer to a struct that needs to be copied from user space. This struct contains the address of the first word in memory to be written to the slave device, the number of words to be transferred, and the data that should be written to the control register of the firmware module.

After extracting this data and saving it, the main loop of this operation begins. This loop is shown in Figure 5.6.

The first step is to determine the length of the next DMA transfer. The maximum length of a DMA transfer is 1024 due to the size of the DMA buffer. If there are 1024 or more words that remain in the total transfer, this is selected as the length of the current transfer. If not, the number of remaining words in the transfer is selected. Once the length of the current transfer has been selected, the actual DMA transfer can begin. Once it has finished, the number of completed words thus far in the transfer is incremented, as is the address of the next data to be written. If the transfer is complete, the loop will break at this point. If not, the JTAG or SelectMAP control register will be written to initiate the transfer in the hardware module (if it has not already been initiated). Finally, the last step in the loop is to sleep until the FIFO half empty interrupt fires. This call is surrounded by a check to the status register of the hardware module to ensure that enough data has been written to clear the interrupt. This is important because the interrupt is rising edge sensitive, meaning that it will only trigger as it goes from low to high. If the process goes to sleep waiting for the interrupt while the interrupt line is high, it will time out because the interrupt will never fire again.

The read operation is similar and uses the same struct for the third parameter as is described above. The main loop of the read operation is shown in Figure 5.7. Before entering this loop, the read transfer is started in the firmware module and the process sleeps until the interrupt indicating that the FIFO has reached half full arrives. In the main loop, the length of the current transfer is computed just as before. Then a DMA read operation is started and the completed length and the destination address are incremented upon completion. If there are still 1024 or more words to be read from the device, the process will sleep until the FIFO halfway mark interrupt fires again. If not, the driver simply polls the status register of the firmware module until the transfer is complete because the FIFO interrupt will not fire again. Once this is complete, it will return to the top of the loop and initiate a DMA transfer to read the remaining words from the device.

5.4.3 User Space Changes

The changes to user space code are relatively minor. A DMA driver class has been created which handles all interactions with the device drivers in the kernel. This includes making IOCTL

```
while(completed_length < total_transfer){
        // Select how many words to transfer
        current_transfer = (completed_length + DMA_READ_TRANSFER_LENGTH >
    total_transfer) ? remainder_words : DMA_READ_TRANSFER_LENGTH;

        //read the next set of words and increment address/completed word
    counter
        //The while loop is to retry if the read fails
        while(jcm_dma_read(working_address, current_transfer, dev_id,
    jcmBaseAddress + JCM_DMA_CONTROL_OFFSET));
        completed_length += current_transfer;
        working_address += current_transfer;

        //At this point, the FIFO interrupt will never fire
        if((total_transfer - completed_length) < JTAG_FIFO_THRESHOLD){
                //Wait for transfer to complete
                while(jcm_is_busy());
                //After this wait, return to the top of loop and read until
    complete
        }
        // If we have read enough data for the interrupt to clear, wait for
    interrupt
        else if(jtag_interrupt_is_cleared()){
                        wait_for_jcm_interrupt();
        }
}
break;
```

Figure 5.7: JTAG Read IOCTL Operation

calls and creating structs to be passed to kernel space. To maintain backwards compatibility in the software with older hardware versions, an object of this class is created only if the current firmware version supports DMA. This happens automatically based on the contents of a version register present in the firmware. When the DMA driver object is used, all interactions with the hardware that occur in the user space code are automatically redirected to the DMA driver object with no user intervention. In this way, high-level user code that makes use of the JCM software library does not need to change when using different firmware versions.

This chapter has described the changes that have been made to the Linux kernel of the JCM. These changes are in the form of device tree nodes and kernel modules which facilitate software interaction with the firmware modules described previously. The performance of this system will be described in the following chapter.

CHAPTER 6. PERFORMANCE

This chapter analyzes the performance of the new JCM system with DMA. First, the data rates of the new system will be compared with the original JCM, followed by a comparison of the processor utilization. This leads into a discussion of the Multi-JTAG system made possible by the more efficient utilization of processor resources.

6.1 Data Rate

All of the data rate measurements in this section were obtained using the same method which was described in Section 3.2.3. The time it takes to transfer a large amount of data (800 Mb for JTAG, 4,000 Mb for SelectMAP) between the JCM and a slave device is measured. Then the size of the transfer is divided by the measured time to get the data rate in Mbps.

A chart comparing the JTAG data rates with and without DMA is shown in Figure 6.1. The differences are hard to see at lower clock speeds because the version without DMA does not have too much trouble keeping up. However, at higher clock speeds the standard version falls well short of the potential maximum data rate. The DMA version, on the other hand, remains very close to the maximum data rate for all clock speeds. The increased speed and efficiency of moving data to the module using DMA means that the processor is able to feed data to the module more quickly, increasing the data rate as expected.

The difference between read and write operations using DMA is also much smaller with writes just barely edging out reads in terms of data rate. This is due to the state machine changes that make it so a read transfer can happen without first filling the write FIFO. This way, no unnecessary data is transferred to the module during a read, which increases the data rate.

The data rates for 8-, 16-, and 32-bit SelectMAP are shown in Figures 6.2, 6.3, and 6.4 respectively. The increased data rate that DMA can provide is even more apparent in the SelectMAP interface because the potential data rate is so much higher. Notice that the standard JCM transfers

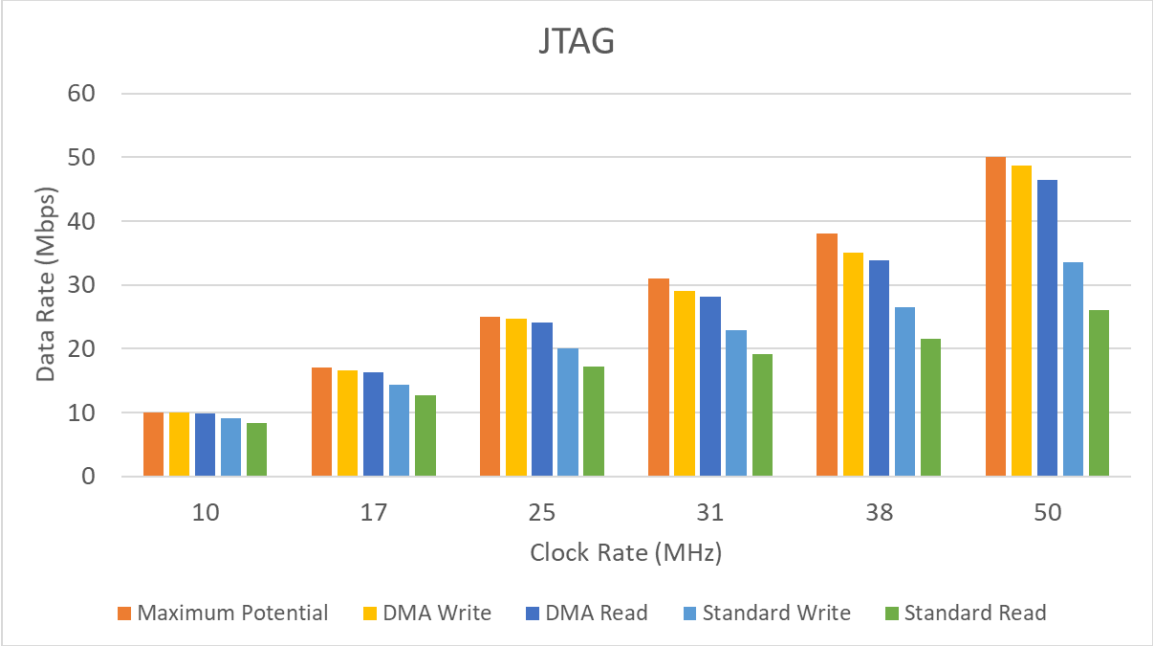

Figure 6.1: JTAG Data Rate Comparison

cannot exceed a data rate of about 100 Mbps. The data rate with DMA, on the other hand, can exceed 950 Mbps in some configurations.

This is a huge improvement over the original design, but it still is not perfect. When using 32-bit mode at high clock speeds, the data rate actually starts to drop compared to lower clock speeds. This appears to be because the driver times out occasionally when waiting for interrupts to arrive. At a 50 MHz clock in 32-bit mode, the speed at which the DMA engine moves data in and out of the hardware module is similar to the speed at which the module moves data in and out of the FIFOs while performing reads and writes using SelectMAP. This appears to cause some glitches on the FIFO half full interrupt line as the number of words in the FIFO hovers around the half full threshold. These glitches are tricky to guard against and lead to some missed interrupts in the driver. The driver recovers using a timeout when waiting for an interrupt, but the extra time it takes to wake up limits the data rate. Understanding this issue more fully and addressing it is an important task for future work.

There is also a clear difference in the data rate of reads and writes at high speeds using SelectMAP in 16-bit or 32-bit mode. The maximum data rate of a write operation is about 950 Mbps, whereas the maximum data rate of a read is only about 550 Mbps. The reason for this

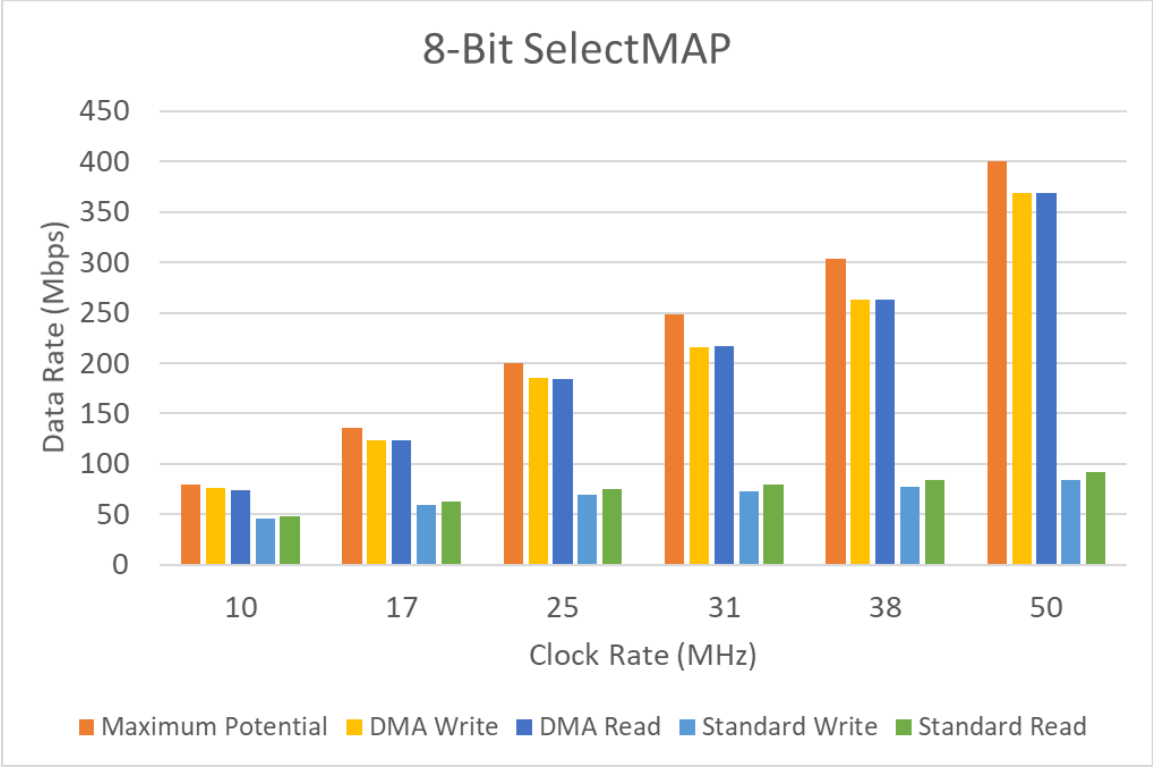

Figure 6.2: 8-Bit SelectMAP Data Rate Comparison

is not clear at this time, but one possibility is that writes to main memory (which would occur when using DMA to read from a module) are given less priority by the memory controller than reads from memory. This could create a bottleneck and extend the time of DMA transfers from the hardware module to memory when there is other memory traffic. Exploring this aspect of the performance is a good opportunity for future work.

6.2 Processor Utilization

The processor utilization during write operations using various interfaces is shown in Figure 6.5. Previously, the processor utilization was greater than 99% whenever the JCM hardware modules were in use. Utilization approaches this for high throughput configurations such as 16- and 32-bit SelectMAP operating at high frequencies. This happens because the user program never gets to sleep for very long before the process wakes up to start new DMA transfers. Even so, the utilization never exceeds 84% when using the SelectMAP interface. This number could potentially be reduced by increasing the size of the FIFO within the SelectMAP module and transferring more

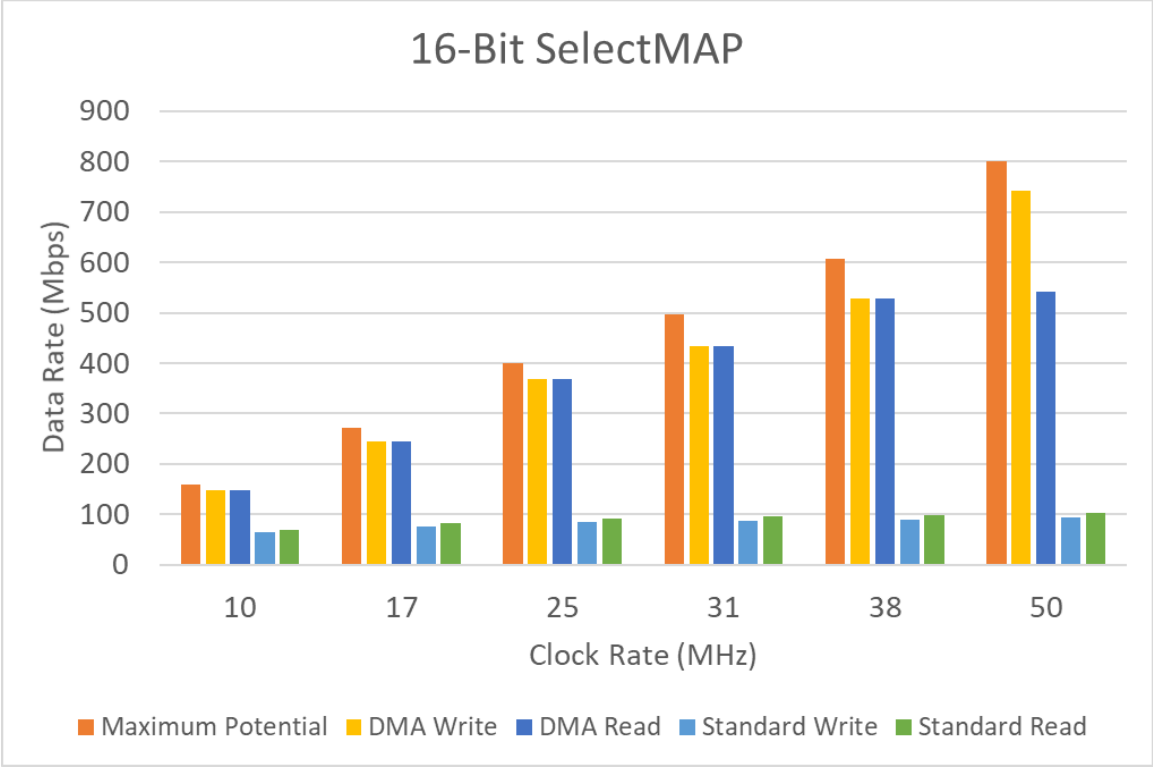

Figure 6.3: 16-Bit SelectMAP Data Rate Comparison

data at a time with DMA. This way, the processor does not have to interact with the hardware as often.

When using a lower throughput interface like JTAG, the utilization can be as low as 1%. Even at its maximum speed, the processor utilization using JTAG never exceeds 6%. This is a huge improvement over the previous system and it leaves the processor open to do a variety of other tasks. The Multi-JTAG system takes advantage of this by managing multiple JTAG chains using the same processor.

6.3 Multi-JTAG

As explained previously, the Multi-JTAG system can manage up to eight JTAG chains simultaneously. The data rates of the system running eight JTAG chains is shown in Table 6.1 and the processor utilization is shown in Figure 6.5. Even while transferring data between eight slave devices simultaneously, the data rate is higher than the data rate of the original JCM system. It trails only slightly behind the improved data rate of the DMA system as shown in the table.

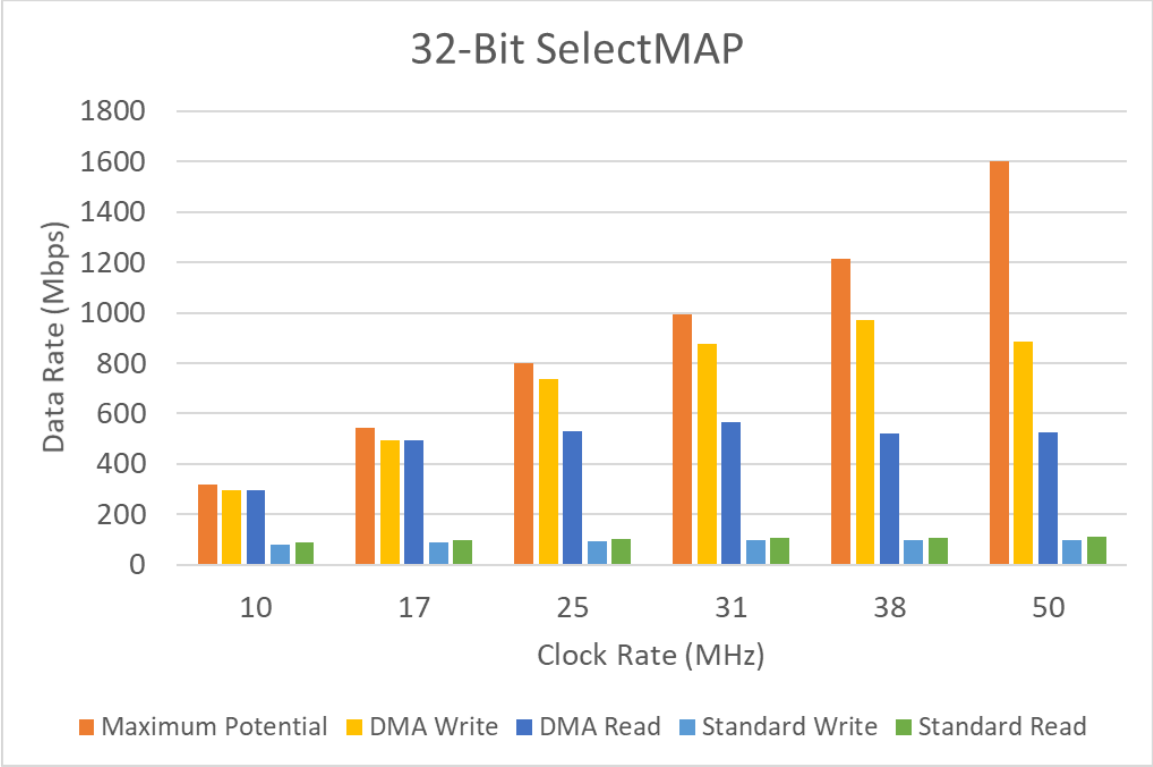

Figure 6.4: 32-Bit SelectMAP Data Rate Comparison

Table 6.1: Multi-JTAG Data Rates

Clock Rate	Data Rate (Single JTAG)	Data Rate Per Port (Multi-JTAG)	Maximum Total Data Rate (Multi-JTAG)	Total Data Rate (Multi-JTAG)
10 MHz	9.96 Mbps	9.72 Mbps	80 Mbps	77.77 Mbps
17 MHz	16.54 Mbps	15.90 Mbps	136 Mbps	127.16 Mbps
25 MHz	24.70 Mbps	23.30 Mbps	200 Mbps	186.37 Mbps
31 MHz	29.03 Mbps	27.12 Mbps	248 Mbps	216.94 Mbps
38 MHz	35.07 Mbps	32.33 Mbps	304 Mbps	258.61 Mbps
50 MHz	48.77 Mbps	43.71 Mbps	400 Mbps	349.65 Mbps

These results are significant because a single system can be used to test up to 8 devices in parallel without significant performance degradation. This allows for much more efficient use of available hardware, potentially relieving a cost burden on JCM users as well as a labor burden on our lab that must employ students to build JCM Breakout Boards and other hardware. The Multi-JTAG system has been deployed on the TURTLE project mentioned in Chapter 2. Researchers on the project use only two JCM systems to inject faults on 10 different boards simultaneously.

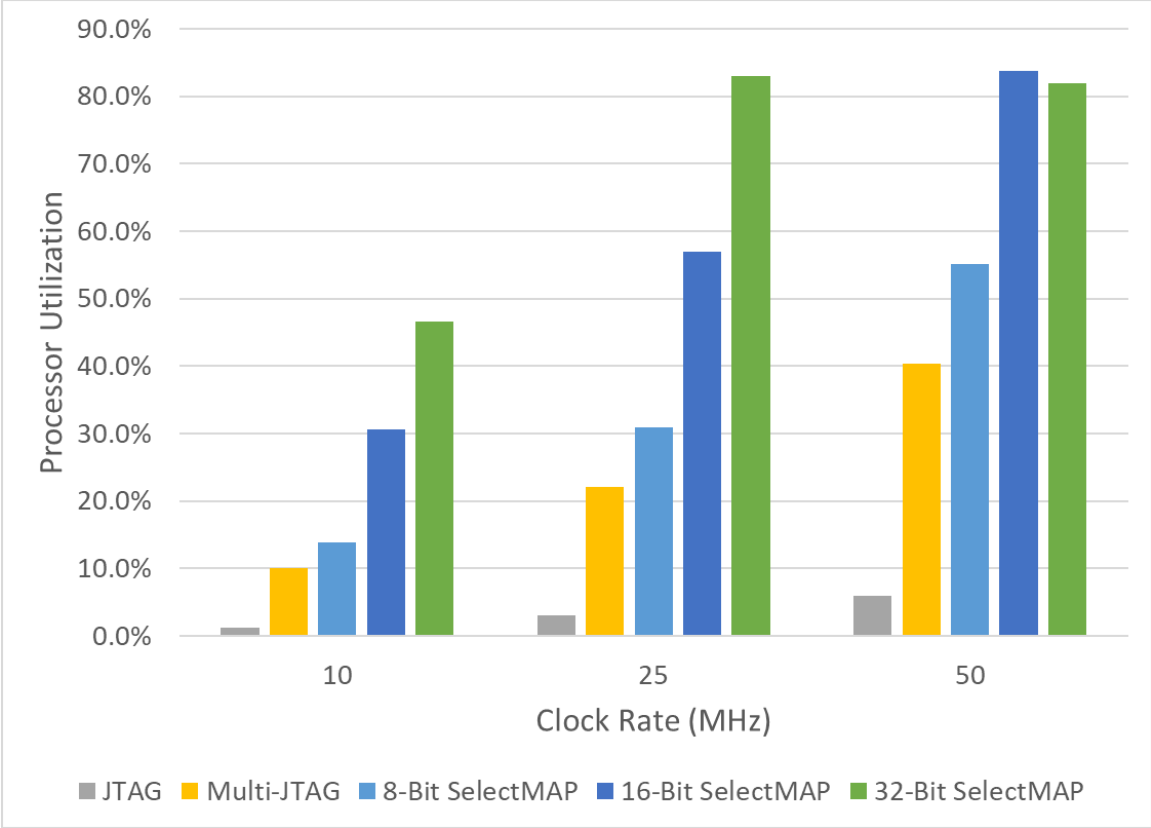

Figure 6.5: Processor Utilization Using DMA

This chapter has presented the performance of the new JCM architecture described in this thesis. The improvements in data rate and utilization provide higher speed configuration management and more efficient hardware usage. There is still room for improvement because the maximum data rates are not achieved at higher clock rates, but it provides much better performance over the previous version of the JCM.

CHAPTER 7. CONCLUSION

The system described in this thesis has improved upon the original JCM by adding DMA and interrupt support. This has resulted in faster data rates (up to 9.97x speedup) using both the JTAG and SelectMAP interfaces. It has also decreased processor utilization dramatically, improving the overall performance and responsiveness of the system. By decreasing processor utilization, the JCM is also able to manage up to eight JTAG chains simultaneously. This multi-JTAG system has been used in the BYU TURTLE project to collect fault injection data from ten FPGAs using only two JCMs.

There are still many opportunities for future work on this system. Some aspects of the performance are still not fully understood and require further investigation to improve. For example, the reason behind the slower upper limit of the data rate of SelectMAP read operations compared to write operations is still unknown and needs to be resolved. The exact factors that limit the data rate of the SelectMAP interface at high speeds are also unknown.

The processor utilization is still very high when the SelectMAP interface operates at high speeds. This could potentially be reduced by increasing the size of the buffer FIFOs and transferring more data at a time via DMA. More experimentation with buffer and transfer sizes could lead to lower utilization and possibly even higher data rates.

The basic functionality of the JCM also has room to expand to incorporate more than just Xilinx FPGAs. The SelectMAP interface is unique to Xilinx devices, but the JTAG interface is used in thousands of other devices. The JCM could be not only an FPGA configuration manager, but also a flexible JTAG tool that can be used to interact with a wide variety of electronics. The JCM software is currently being revised to further decouple basic JTAG functionality from Xilinx-specific functionality. This will allow for more flexible JTAG usage in various situations.

The user interface to the JCM presents more opportunities for improvements. A client-server model is under development that will replace the current model in which user-created pro-

grams are run natively on the JCM. With the new model, the JCM will simply run a server program that accepts requests from clients. User programs will consist of Python scripts that open a connection to the JCM server and then send a series of requests to the server. This model will make it easier to utilize the multiple JTAG ports by letting the server take care of executing requests in parallel. It will also make rapid development and scripting simpler for users and remove the need for compilation on the JCM.

The JCM has been an essential part of many of the research activities of the BYU Configurable Computing Lab. The need for high-speed configuration management will only increase as larger and more complex devices are released. The improvements to the JCM described in this thesis make it even more capable of filling that need as a programmable, high-speed configuration manager.

REFERENCES

[1] I. Kuon, R. Tessier, and J. Rose, "FPGA Architecture: Survey and Challenges," *Foundations and Trends® in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2007. [Online]. Available: http://www.nowpublishers.com/article/Details/EDA-005 1, 3

[2] A. Putnam, G. Jan, G. Michael, H. S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. M. Caulfield, A. Smith, J. Thong, P. Yi, X. D. Burger, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Prashanth, A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 13–24, oct 2014. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2678373.2665678 1

[3] N. Montealegre, D. Merodio, A. Fernndez, and P. Armbruster, "In-flight reconfigurable FPGA-based space systems," in *Adaptive Hardware and Systems (AHS), 2015 NASA/ESA Conference on*, June 2015, pp. 1–8. 1, 4

[4] C. Maunder, "The joint test action group," *Computer-Aided Engineering Journal*, vol. 3, no. 4, pp. 121–122, August 1986. 1, 7

[5] A. Gruwell, P. Zabriskie, and M. Wirthlin, "High-speed FPGA configuration and testing through JTAG," in *2016 IEEE AUTOTESTCON*. IEEE, sep 2016, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/7589601/ 2, 14, 15

[6] B. Ronak and S. A. Fahmy, "Mapping for Maximum Performance on FPGA DSP Blocks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 4, pp. 573–585, apr 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7229289/ 3

[7] Xilinx Inc, "Xilinx UG071 Virtex-4 FPGA Configuration Guide, User Guide," p. 87. [Online]. Available: https://www.xilinx.com/support/documentation/user{_}guides/ug071.pdf 3

[8] ——, "Virtex-5 FPGA Configuration User Guide (UG191)," p. 19. [Online]. Available: https://www.xilinx.com/support/documentation/user{_}guides/ug191.pdf 3

[9] ——, "Virtex-6 FPGA Configuration User Guide (UG360)," p. 88. [Online]. Available: https://www.xilinx.com/support/documentation/user{_}guides/ug360.pdf 3

[10] ——, "7 Series FPGAs Configuration User Guide (UG470)," pp. 40–47. [Online]. Available: https://www.xilinx.com/support/documentation/user{_}guides/ug470{_}7Series{_}Config.pdf 3, 4, 5, 8, 10, 11, 18, 21, 23

[11] ——, "UltraScale Architecture Configuration User Guide (UG570)," pp. 20–21. [Online]. Available: https://www.xilinx.com/support/documentation/user{_}guides/ug570-ultrascale-configuration.pdf 3

[12] A. B. Gruwell, "BYU ScholarsArchive High-Speed Programmable FPGA Configuration Memory Access Using JTAG," Thesis, Brigham Young University, 2017. [Online]. Available: https://scholarsarchive.byu.edu/etd/6321 8, 15, 16

[13] Xilinx Inc, "Kintex-7 FPGAs Data Sheet: DC and AC Switching Characteristics (DS182)," p. 65, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/data{_}sheets/ds182{_}Kintex{_}7{_}Data{_}Sheet.pdf 9

[14] ——, "Partial Reconfiguration User Guide," 2013. [Online]. Available: https://www.xilinx.com/support/documentation/sw{_}manuals/xilinx14{_}7/ug702.pdf 11

[15] H. Michel, A. Belger, T. Lange, B. Fiethe, and H. Michalik, "Read back scrubbing for SRAM FPGAs in a data processing unit for space instruments," in *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, jun 2015, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/7231149/ 11, 13

[16] L. Antoni, R. Leveugle, and B. Feher, "Using run-time reconfiguration for fault injection applications," *IEEE Transactions on Instrumentation and Measurement*, vol. 52, no. 5, pp. 1468–1473, Oct 2003. 11

[17] E. Fuller, M. Caffrey, P. Blain, C. Carmichael, N. Khalsa, and A. Salazar, "Radiation test results of the virtex fpga and zbt sram for space based reconfigurable computing," in *MAPLD proceedings*, vol. 2, 1999. 11

[18] M. Wirthlin, A. M. Keller, C. McCloskey, P. Ridd, D. Lee, and J. Draper, "SEU Mitigation and Validation of the LEON3 Soft Processor Using Triple Modular Redundancy for Space Processing," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '16*. New York, New York, USA: ACM Press, 2016, pp. 205–214. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2847263.2847278 12

[19] D. S. Lee, G. R. Allen, G. Swift, M. Cannon, M. Wirthlin, J. S. George, R. Koga, and K. Huey, "Single-Event Characterization of the 20 nm Xilinx Kintex UltraScale Field-Programmable Gate Array under Heavy Ion Irradiation," in *2015 IEEE Radiation Effects Data Workshop (REDW)*. IEEE, jul 2015, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/7336736/ 12

[20] M. Wirthlin, M. Citterio, C. Meroni, M. Cannon, and A. Camplani, "Evaluating Multi-Gigabit Transceivers (MGT) for Use in High Energy Physics Through Proton Irradiation." feb 2015. [Online]. Available: http://cds.cern.ch/record/1995019 12

[21] F. Duhem, F. Muller, and P. Lorenzini, "Farm: Fast reconfiguration manager for reducing reconfiguration time overhead on fpga," in *Reconfigurable Computing: Architectures, Tools*

and Applications, A. Koch, R. Krishnamurthy, J. McAllister, R. Woods, and T. El-Ghazawi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 253–260. 13

[22] K. Vipin and S. A. Fahmy, "ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 41–44, sep 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6780588/ 13

[23] J. Tonfat, F. Kastensmidt, and R. Reis, "Energy efficient frame-level redundancy scrubbing technique for SRAM-based FPGAs," in *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, jun 2015, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/7231160/ 13

[24] A. Stoddard, A. Gruwell, P. Zabriskie, and M. Wirthlin, "A Hybrid Approach to FPGA Configuration Scrubbing," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 497–503, jan 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7776929/ 13

[25] Xilinx Inc, "Vivado Design Suite Tcl Command Reference Guide," 2014. [Online]. Available: https://www.xilinx.com/support/documentation/sw{_}manuals/xilinx2014{_}4/ug835-vivado-tcl-commands.pdf 13

[26] ——, "AXI Reference Guide," p. 9. [Online]. Available: https://www.xilinx.com/support/documentation/ip{_}documentation/ug761{_}axi{_}reference{_}guide.pdf 29, 33

[27] ——, "AXI DMA v7.1 LogiCORE IP Product Guide (PG021)," pp. 12–13. [Online]. Available: https://www.xilinx.com/support/documentation/ip{_}documentation/axi{_}dma/v7{_}1/pg021{_}axi{_}dma.pdf 35

[28] P. J. Salzman, M. Burian, O. Pomerantz, P. J. Salzman, M. Burian, and O. Pomerantz, "The Linux Kernel Module Programming Guide," 2007. [Online]. Available: http://lib.hpu.edu.vn/handle/123456789/21420 42, 45

[29] G. Likely and J. Boyer, "A Symphony of Flavours: Using the device tree to describe embedded hardware," *Embedded Linux Conference*, vol. 2, pp. 27–37, 2008. 44

[30] Xilinx Inc, "Xilinx Wiki - Build Device Tree Blob." [Online]. Available: http://www.wiki.xilinx.com/Build+Device+Tree+Blob 45