High-Speed Programmable FPGA Configuration Memory Access Using JTAG

Ammon Bradley Gruwell

Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Electrical and Computer Engineering Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/6321

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
High-Speed Programmable FPGA Configuration Memory Access Using JTAG

Ammon Bradley Gruwell

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
James K. Archibald
Doran K. Wilde

Department of Electrical and Computer Engineering
Brigham Young University

Copyright © 2017 Ammon Bradley Gruwell
All Rights Reserved
ABSTRACT

High-Speed Programmable FPGA Configuration Memory Access Using JTAG

Ammon Bradley Gruwell
Department of Electrical and Computer Engineering, BYU
Master of Science

Over the past couple of decades Field Programmable Gate Arrays (FPGAs) have become increasingly useful in a variety of domains. This is due to their low cost and flexibility compared to custom ASICs. This increasing interest in FPGAs has driven the need for tools that both qualify and improve the reliability of FPGAs for applications where the reconfigurability of FPGAs makes them vulnerable to radiation upsets such as in aerospace environments. Such tools ideally work with a wide variety of devices, are highly programmable but simple to use, and perform tasks at relatively high speeds. Of the various FPGA configuration interfaces available, the Joint Test Action Group (JTAG) standard for serial communication is the most universally compatible interface due to its use for verifying integrated circuits and testing printed circuit board connectivity. This universality makes it a good interface for tools seeking to access FPGA configuration memory.

This thesis introduces a new tool architecture for high-speed, programmable JTAG access to FPGA configuration memory. This tool, called the JTAG Configuration Manager (JCM), is made up of a large C++ software library that runs on an embedded micro-processor coupled with a hardware JTAG controller module implemented in programmable logic. The JCM software library allows for the development of custom JTAG communication of any kind, although this thesis focuses on applications related to FPGA reliability. The JCM hardware controller module allows these software-generated JTAG sequences to be streamed out at very high speeds. Together the software and hardware provide the high-speed and programmability that is important for many JTAG applications.

Keywords: FPGA, JTAG, scrubbing, fault injection, radiation testing, laser testing, C++, Python, BYU, configuration, readback, reliability, SEU, Xilinx, bitstream, upset mitigation, hardware, software, high speed, configuration memory, CRAM, SRAM, Joint Test Action Group, ionizing radiation, space, aerospace, interface, serial, ARM processor, architecture, programmable logic, MicroZed, Zynq, Linux nonomenclature
ACKNOWLEDGMENTS

First I’d like to thank my graduate advisor Dr. Michael J. Wirthlin for the vision and direction that helped this project become a great success. There were various occasions that I had doubts or questions about the direction this work should take; however, a quick conversation with him always helped me overcome these obstacles. I’m also grateful for the many hours he put into improving and refocusing my writing despite his relentlessly busy schedule. I appreciate that his sound advice and criticism always came with the comment that it was my thesis and thus ultimately my decision. I believe that the critical thinking that came with this helped me become a better writer and researcher.

My gratitude also goes out to my graduate committee, Dr. Doran K. Wilde and Dr. James K. Archibald, and the members of the BYU Configurable Computing Lab who contributed to the JCM project. Peter Zabriskie made huge improvements to the JCM hardware and software, and Andy Wilson is responsible for all JCM-processor interaction. Thanks also to Luke Newmeyer and Kyle Davidson for their work in developing the JTAG breakout board.

I’d also like to thank my wife Jamie for her endless support and patience throughout this long process. She motivated me to put in the long hours necessary to make this thesis a reality. This accomplishment is as much hers as it is mine. While I was away from home researching and writing, she was laboring at home to care for our two young children. I never would have finished without her patience and encouragement.

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).
Contents

List of Tables vii

List of Figures viii

1 Introduction 1

2 JTAG 4

2.1 The JTAG Protocol . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.2 TAP State Machine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.3 Signals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.4 Daisy Chaining . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.5 JTAG and Xilinx FPGAs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

3 The JTAG Configuration Manager 11

3.1 The Avnet MicroZed Board . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.2 The Xilinx Zynq SoC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.3 The Custom JTAG Carrier Card . . . . . . . . . . . . . . . . . . . . . . . . 14

4 The JTAG Hardware Controller 16

4.1 JTAG Signal Generation and Sampling . . . . . . . . . . . . . . . . . . . . . 16
  4.1.1 The Software Approach . . . . . . . . . . . . . . . . . . . . . . . . . . 16
  4.1.2 The Hardware Approach . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.2 The JTAG Hardware Controller ............................................. 17
4.2.1 The JTAG Hardware Controller Registers .............................. 18
4.2.2 The JTAG Hardware Controller State Machine ....................... 20
4.2.3 The JTAG Hardware Controller System Integration .................. 22
4.3 Speed Calibration .............................................................. 24

5 The JCM Software Library ..................................................... 28
5.1 Software Architecture ......................................................... 28
5.1.1 Firmware Level .............................................................. 29
5.1.2 JTAG Level ................................................................. 30
5.1.3 Device Level ................................................................. 31
5.1.4 User API Level .............................................................. 34
5.2 Scripting Interface .............................................................. 36

6 JCM Applications ................................................................. 38
6.1 FPGA Configuration ............................................................ 38
6.2 Configuration Scrubbing ....................................................... 39
6.2.1 Blind Scrubbing ............................................................. 40
6.2.2 Readback Scrubbing ....................................................... 40
6.2.3 Hybrid Scrubbing ........................................................... 42
6.2.4 Radiation Testing with the JCM ........................................ 44
6.3 Fault Injection ................................................................. 45
6.4 Processor Debugging ......................................................... 50
6.5 Design Monitoring and Debugging ........................................ 51
6.6 Temperature and Voltage Monitoring .................................... 51
6.7 Remote Tool Control ......................................................... 52
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Xilinx 7-Series JTAG Registers Used by the JCM</td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>The JCM Hardware Control Register Bits</td>
<td>21</td>
</tr>
<tr>
<td>4.2</td>
<td>The JCM Hardware Status Register Bits</td>
<td>22</td>
</tr>
<tr>
<td>4.3</td>
<td>JTAG Hardware Controller Resource Utilization</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Firmware-Level Library Functions</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>JTAG-Level Library Functions</td>
<td>32</td>
</tr>
<tr>
<td>5.3</td>
<td>Device-Level Library Functions</td>
<td>35</td>
</tr>
<tr>
<td>5.4</td>
<td>User API Library Functions</td>
<td>37</td>
</tr>
<tr>
<td>6.1</td>
<td>JCM Function Delays on a Kintex 7 FPGA</td>
<td>41</td>
</tr>
<tr>
<td>A.1</td>
<td>Bits of a Xilinx 7-Series Frame Address [1]</td>
<td>62</td>
</tr>
<tr>
<td>A.2</td>
<td>Type 1 Configuration Packet Format [1]</td>
<td>63</td>
</tr>
<tr>
<td>A.3</td>
<td>Type 2 Configuration Packet Format [1]</td>
<td>63</td>
</tr>
<tr>
<td>A.4</td>
<td>A Command Sequence to Write to Configuration Memory</td>
<td>64</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Test Access Port State Machine [2] ................................................... 6
2.2 JTAG Chaining .............................................................................. 7
2.3 Example JTAG Waveform ............................................................ 8

3.1 System Block Diagram ................................................................. 12
3.2 The JCM Hardware .................................................................. 13

4.1 JTAG Controller State Machine ................................................. 18
4.2 Sequence of JTAG Hardware Controller Register Access .......... 20
4.3 JTAG Transaction Waveform ..................................................... 21
4.4 JCM Block Diagram ................................................................. 23
4.5 Radiation Testing with Long JTAG Cables ................................. 24
4.6 Using An LVDS Cable With The JCM ....................................... 25
4.7 Timing Delay Fixed By Dynamic Calibration ............................. 26

5.1 The JCM Software Layers of Abstraction ................................. 29
5.2 Resetting the JTAG State Machine With Firmware-Level Commands .... 31
5.3 Reading a Register with JTAG-level Commands .................... 33
5.4 A JCM Configuration File For A Xilinx Kintex-7 FPGA .......... 34
5.5 Example Configuration Sequence Using Device-Level Commands .... 34
5.6 Commands of a Typical JCM Session ...................................... 36
Chapter 1

Introduction

Field Programmable Gate Arrays (FPGAs) are configurable integrated circuits that have become increasingly common in recent years due in part to their low cost, fast development cycle, and re-programmability compared to traditional ASIC solutions [4]. These and other attributes make them very useful in domains that require a small number of high-performing yet flexible units such as aerospace. FPGAs are especially beneficial in space applications, such as satellites in orbit, because of their low cost and their ability to be remotely adapted and configured, allowing for functional flexibility despite physical inaccessibility [5] [6].

For various applications, the ability to access and modify FPGA state information is very useful. For example, using FPGAs reliably in aerospace applications can be difficult due to the presence of high levels of ionizing radiation [7]. This radiation is problematic because the state information of FPGAs is typically stored in Static Random-Access Memory (SRAM), a memory technology which is vulnerable to radiation upsets [8]. Such upsets occur when ionized particles strike the SRAM memory cells containing FPGA state information and cause the memory values to change.

This upset vulnerability among other reasons motivates the need for tools that perform error mitigation and reliability qualification. These operations require high-speed programmable access to FPGA configuration memory (CRAM). This high-speed CRAM access takes various forms including: configuring the FPGA, querying the CRAM (readback), repairing configuration upsets (scrubbing), artificially changing the state of the CRAM (fault injection), and reading the internal state of the circuit. Performing these operations at a higher speed often increases the effectiveness of applications that depend upon them.
FPGAs typically support a number of interfaces for CRAM access including the Joint Test Action Group (JTAG) standard, SelectMAP, the Internal Configuration Access Port (ICAP), and the Processor Configuration Access Port (PCAP) [9]. Some of these interfaces have higher bandwidth and thus higher speed, while others are designed for lower overhead and flexibility. Choosing the correct interface depends upon specific requirements and needs.

Of the various interfaces, JTAG is well suited to meet the needs of tools requiring universal and programmable FPGA CRAM access. It is the industry standard for verifying integrated circuits and testing circuit board connectivity, and has become the most universally supported FPGA configuration interface [10]. Its wide adoption is due primarily to its low overhead, flexibility, and scalability. The low-overhead comes from needing to support only four pins. Despite this small footprint, JTAG provides the flexibility to support communication with any number of registers on any number of devices chained together. One downside to using so few pins is a slower maximum speed compared to some other high-bandwidth parallel interfaces. This motivates the need for a maximization of JTAG speed to compensate for its few data lines.

JTAG’s benefits have made it the default interface for FPGA configuration. This makes it a good interface for tools that need to interact with a variety of devices, since the same JTAG port used for simple reconfiguration can be used for more complex applications that require CRAM access. While many tools support simple JTAG functions at relatively high speeds, custom JTAG sequences are typically created using scripting languages such as TCL and are limited by the slow speed of script interpreters.

This thesis will demonstrate a novel FPGA-based system that provides both high-speed JTAG communication and software programmability. This system is called the JTAG Configuration Manager (JCM). It features a software library of FPGA applications running on an embedded ARM processor. A simple software API provides a highly-programmable interface for creating custom JTAG sequences that access FPGA CRAM. Coupled tightly with the software is an FPGA hardware peripheral that generates the high-speed JTAG signals. This hardware signal generation is much faster and less processor intensive than it would be in software, allowing for a maximum JTAG bandwidth. The main contributions of this thesis are the description of a hardware design that maximizes the speed of JTAG signal
generation and sampling and the presentation of a unique architecture combining the speed of hardware with the programmability of software for generating custom JTAG sequences. Some of the work in this thesis has been published previously in [11] and [12].

Although the JTAG Configuration Manager could be used for any type of JTAG interaction, it has primarily been used for FPGA reliability applications. Various academic, industry, and government researchers use the JTAG interface and could therefore benefit from the high speed and programmability that the JCM offers [13] [10] [14]. Applications include soft processor debugging, reliability qualification through fault injection, CRAM monitoring during radiation and laser testing, and remote CRAM access over a network.

Tools similar to the JCM exist and are widely used for many of these same applications; however, the JCM has proven successful by combining very high speeds, ease of programmability, and a very low cost of materials compared to industry JTAG controllers. The success of the JCM is manifested in the wide variety of researchers who have put it to use in the short time since its creation.

This thesis begins with a chapter describing JTAG and how it is used to interact with integrated circuits. This chapter discusses in detail how a JTAG transaction takes place, how devices are chained together, and how JTAG is typically used. It then describes several alternative interfaces that can be used to interact with Xilinx FPGAs. Chapter 3 provides an overview of the JTAG Configuration Manager including a description of its physical components. Chapter 4 begins a comprehensive description of the JCM, starting with the JTAG hardware controller circuit implemented in programmable logic. The attributes described in this chapter are the key to maximizing the speed of JTAG. Chapter 5 continues the description of the JCM by introducing the software library. This chapter breaks the architecture down into levels of abstraction with example commands at each level. A description of scripting using the JCM software library is also included. Chapters 6 introduces the main JCM applications. It begins by describing radiation testing and scrubbing. This application is the primary motivation for the JCM. The chapter then covers various other JCM applications such as fault injection and processor debugging. Finally, chapter 7 concludes this work.
Chapter 2

JTAG

The Joint Test Action Group (JTAG) interface is a serial communication standard that is used for verifying integrated circuits and testing circuit board connectivity. This chapter introduces JTAG and how it is used for a variety of applications. Since JTAG is central to this thesis, understanding the JTAG protocol is key to comprehending the advantages of the novel architecture this thesis describes. Despite this and subsequent chapters’ focus on using JTAG with FPGAs as an example application, the work of this thesis is pertinent to a wide variety of other JTAG applications.

2.1 The JTAG Protocol

JTAG is an IEEE standard (1149.1) that specifies a dedicated debug port that interfaces with an on-chip finite state machine called the Test Access Port (TAP). Using this state machine, JTAG provides access to any number of test registers and device sub-circuits. It has become the industry standard for verifying integrated circuits and debugging embedded systems. Much of the material of this section is drawn from the IEEE standard specification [15].

JTAG is used for a wide variety of testing and debugging applications both at the device level as well as the system level. As board dimensions shrink and surface-mount packaging technology improves, the old way of testing circuits with test probes becomes increasingly effective. Instead, JTAG is used as a low-cost interface to access any internal circuit signal without the need for physical test points [16].

One of the most common uses for JTAG is testing the sub-blocks and connections of integrated circuits. Its four-pin interface is a low-cost means of exposing the internal signals of embedded systems for purposes of verification and debugging. Adding extra pins
and internal circuitry for debugging purposes is a costly but necessary overhead expense to circuit designers, so limiting that interface to four pins is ideal. After a chip has been manufactured, JTAG is then used to identify a variety of manufacturing defects such as poor connections, short circuits, and stuck bits.

JTAG is also commonly used for programming the CRAM of FPGAs. Since it is already used for the testing and verification of manufactured FPGAs and thus has access to the internal circuitry of every FPGA, it is a natural choice as the interface to be used for setting the initial values of CRAM as well as performing dynamic partial reconfiguration during circuit operation. The use of JTAG for accessing the configuration registers of Xilinx FPGAs will be discussed in greater detail in the following chapter.

2.2 TAP State Machine

JTAG is a flexible interface enabling it to communicate with any number of different entities. It utilizes two main registers that are controlled by the TAP state machine: the Instruction Register (IR) and the Data Register (DR). The DR is merely a portal that maps to any one of the various entities that are accessible with JTAG. The IR holds an address that specifies to which of the available entities the DR should map. For example, when the IR contains the ID code register address, the device’s ID code can be read out of the JTAG DR, but when the IR contains the Bypass register address, the DR contains only a 1-bit '0' value.

The JTAG Test Access Port (TAP) is a simple state machine that manages all interaction with the IR and DR. It is what enables JTAG to be flexible and scalable despite only having one control signal. The state machine, shown in Figure 2.1, consists of 16 states that can be divided into three different sections: initialization, IR communication, and DR communication. The initialization section includes a reset state (TLR), which is typically used in between transactions, and an idle state (RTI), which can be used in between parts of a single transaction. The IR communication section contains several states that handle communication with the IR. Once the IR has been loaded with the address of a JTAG destination register, the DR communication section of the TAP state machine is used to read and write data from the specified target.
The available JTAG destination registers depend upon the slave device. The IEEE standard requires that a few mandatory registers be supported such as Boundary Scan (BSR) and BYPASS; however, the Test Access Port (TAP) implementor is free to add on additional registers as desired. Since the IR can be any length, any number of different JTAG destination registers can be supported, allowing JTAG to scale to meet the needs of any system. This attribute is one of the reasons why JTAG is so universally used.

2.3 Signals

The JTAG protocol requires four serial signals: TDI (Test Data In), TDO (Test Data Out), TCK (Test Clock), and TMS (Test Mode Select), with an optional fifth signal, TRST (Test Reset). These signals can be seen in Figure 2.2 communicating with three different devices on a single chain with TMS and TCK in parallel, but TDI and TDO tied together in series. The JTAG master, which for this work is the JTAG Configuration Manager, controls TMS, TCK, and TDI, while the JTAG slave (typically an FPGA) controls TDO.
This subsection will discuss each of these signals in detail, while a discussion about how devices are chained together is found in the next subsection.

![JTAG Chaining Diagram](Image)

The TMS signal is the only control signal of the four JTAG pins and is driven by the JTAG master. Typical JTAG sequences consist of a number of control (TMS) bits, then the actual data, and finally some additional control bits. These control signals are used to navigate the TAP state machine to one of the states where data can be shifted in or out via TDI and TDO.

The TDI signal is used to transmit all data originating from the JTAG master. Besides being used to feed the JTAG destination register address to the IR, it is also used to both send and request data from the JTAG slave device. The slave then uses the TDO signal to return any data that has been requested.

All of these operations are synchronized by the JTAG clock signal TCK which is also driven by the JTAG master. This clock need not run at a consistent frequency, but enables the master to communicate to the slave when the other signals should be sampled. Its speed is typically limited by the JTAG cable length or the specifications of the slave device to between 1 and 100 MHz. A timing diagram of a typical JTAG transaction is shown in Figure 2.3.
The JTAG reset signal (TRST) can be used by the master to reset the state machine, although five ticks of TCK with TMS held high also results in the state machine returning to the reset state. For this reason TRST is optional. The use of this signal would only be desirable in cases where the need for slightly higher speed can justify the cost of an extra pin.

Since JTAG only uses four signals to communicate, transactions require a relatively large amount of standardized control overhead to give meaning to data that is exchanged. For example, a single data transaction typically requires around 15-25 control bits which, if the data itself is only 32 bits, can be a significant percentage of the total transaction size. However, this low pin-out makes it cheap to thread JTAG signals throughout an IC design for easier verification and debugging. For many applications, JTAG’s low cost far outweighs the extra control overhead.

### 2.4 Daisy Chaining

In many cases it is desirable to connect multiple devices together into a single JTAG chain. This chaining, as seen in Figure 2.2, is accomplished by connecting the TCK and TMS pins of all devices together in parallel, and the TDO signal of each device connected to the TDI signal of the next device on the chain. Then, in order to interact with a single target device, all other devices on the chain are put into BYPASS, while the target device is given some other IR code. This results in data easily being passed to and from the target device with at most a few BYPASS bits present before and/or after the target device data. Figure 2.2 shows the example of the IDCODE address being shifted into the IR of Device 2.
while BYPASS is placed in Device 1 and 3. This allows the ID code of Device 2 to be shifted out with an extra zero digit on each end corresponding to Device 1 and 3.

This chaining of devices has both benefits as well as drawbacks. The primary benefit to chaining is that it eliminates the need for multiple JTAG ports on the same board. It also gives a single JTAG controller access to all devices on the chain. The main drawbacks are the small overhead required for each transaction and the fact that TCK can only run as fast as the chain’s slowest device can support.

2.5 JTAG and Xilinx FPGAs

FPGAs made by Xilinx use JTAG as their primary configuration and debugging interface. This is made possible through a number of JTAG destination registers. Two such registers are CONFIG_IN and CONFIG_OUT. These JTAG destination registers enable communication with the Xilinx Configuration Registers which control and maintain all internal state information. Almost all JTAG sequences generated by the JCM consist of commands and data passed in and out of these two registers. A full list of 7-Series JTAG destination registers used by the JCM can be found in Table 2.1. For more information regarding JTAG interaction with the Xilinx Configuration Registers, see Appendix A.

Table 2.1: Xilinx 7-Series JTAG Registers Used by the JCM

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDCODE</td>
<td>001001</td>
<td>Used to certify that communication is working</td>
</tr>
<tr>
<td>BYPASS</td>
<td>111111</td>
<td>Used when a different device on the chain is active</td>
</tr>
<tr>
<td>ISC_NOOP</td>
<td>010100</td>
<td>No operation, used to clear the Instruction Register</td>
</tr>
<tr>
<td>CFG_OUT</td>
<td>000100</td>
<td>Used to read data out of the Configuration Registers</td>
</tr>
<tr>
<td>CFG_IN</td>
<td>000101</td>
<td>Used to write data to the Configuration Registers</td>
</tr>
<tr>
<td>JPROGRAM</td>
<td>001011</td>
<td>Used to reset the configuration before programming</td>
</tr>
<tr>
<td>USER1</td>
<td>000010</td>
<td>Used for reading/writing to internal design signals</td>
</tr>
<tr>
<td>USER2</td>
<td>000011</td>
<td>Used for reading/writing to internal design signals</td>
</tr>
<tr>
<td>USER3</td>
<td>100010</td>
<td>Used for reading/writing to internal design signals</td>
</tr>
<tr>
<td>USER4</td>
<td>100011</td>
<td>Used for reading/writing to internal design signals</td>
</tr>
<tr>
<td>XADC_DRP</td>
<td>110111</td>
<td>Used for reading on-chip temperature and voltages</td>
</tr>
</tbody>
</table>
This chapter should have impressed upon the reader the inherent difficulties in the creation of custom JTAG sequences. JTAG itself is a complex protocol that condenses the many control and data lines of other parallel interfaces into just four signals. Beyond the difficulty of navigating the state machine are other obstacles such as reading and writing data with correct endianness and handling JTAG chains with multiple targets. Then, after JTAG is understood, Xilinx FPGAs have their own communication interface that must be followed. All of these challenges make correct JTAG sequence creation a formidable task. This all motivates the value of a tool that encapsulates this complexity into a layered software library that provides a simple parameterizable interface for creating JTAG sequences.
Chapter 3

The JTAG Configuration Manager

The central contribution of this thesis is the introduction of a new architecture for creating and generating custom JTAG sequences at high speed. This tool is called the JTAG Configuration Manager (JCM), and is the focus of the next few chapters. This chapter describes how the unique combination of both a low-level hardware state machine and a high-level Linux system work together to provide these benefits. Although either one would be useful on its own, the two combine to provide a platform uniquely well-adapted to benefit applications requiring high-speed programmable JTAG sequence creation.

This new architecture is successful because it allows for user interaction at a high-level without hindering the speed of the underlying hardware. If this tool was implemented completely as a hardware circuit, any modification to the design would require not only hardware design expertise, but also the relatively long amount of time inherent to any hardware design modification. Since JTAG is a very flexible and scalable interface, this tool ought to be just as flexible and easily modifiable. The JCM accomplishes this without sacrificing speed by minimizing the functionality implemented in hardware to those aspects which are necessary for maximizing speed but remain relatively constant for all JTAG applications. The rest is implemented in software so that frequent changes to a design are fast and easy.

The JTAG Configuration Manager (JCM) accomplishes this using a platform that couples an application processor with FPGA fabric. It is composed of two parts: a software library running in Linux on an ARM processor, and a JTAG controller circuit implemented in FPGA fabric. Using this platform, a custom hardware circuit can serve as a processor peripheral that accelerates the reading and writing of JTAG signals. In this way the JCM combines the speed of hardware with the flexibility of software to allow users to create straightforward high-level scripts that result in high-speed JTAG transactions being sent
to the target FPGA. Figure 3.1 shows a diagram of how the JCM interacts with a host computer and a target board.

![System Block Diagram](image)

**Figure 3.1: System Block Diagram**

The JCM hardware platform was designed for low cost, flexibility, and portability. It consists of one board containing the processor and FPGA fabric, and another that breaks out the JTAG signals and performs power regulation. As shown in Figure 3.2, the first board is an Avnet MicroZed board, while the second is a custom daughter card made for the JCM. These two boards paired together make up a self-contained tool that is portable, and easy to integrate into any system.

### 3.1 The Avnet MicroZed Board

The MicroZed was the chosen platform due primarily to its use of a Xilinx Zynq 7000 SoC (which is discussed further in Section 3.2) as well as its small form factor and low cost. It was designed to be either a standalone evaluation board or combined with a carrier card for use in a larger system [17]. To connect to a carrier card, the MicroZed contains two MicroHeaders with a combined 108 user I/O pins. Two of the Zynq FPGA I/O banks connect to these MicroHeaders, allowing for 100 programmable logic I/O pins to be broken out as desired.
The MicroZed contains several other important features including both an Ethernet port as well as a USB-UART port for external communication, and a microSD card for external storage. For the JCM, the attached Micro SD card contains a Linux image with the JCM software library as well as the programming bit file for the FPGA. When the MicroZed is powered up the processor proceeds to boot Linux and then programs the FPGA.

### 3.2 The Xilinx Zynq SoC

The Zynq SoC (XC7Z010) combines a dual-core ARM Cortex-A9 processor with 28nm Artix-7 based FPGA fabric [18]. This combination of a hard processor with an FPGA made the Zynq an ideal choice for implementing the new architecture described above. While the processor allows for software flexibility and scalability, the programmable logic makes high speeds possible.
The JCM software library runs on the embedded ARM processor running a light-weight distribution of Linux called Arch-Linux. The processor boots from a micro-SD card that stores the Linux image. Along with Linux, the micro-SD card holds the JCM software library and the FPGA bitstream which contains the JTAG controller module. As the processor boots, the FPGA is programmed. After the boot process completes, the user can login and proceed to run JCM applications from the terminal. The Linux distribution, although relatively light-weight, is still equipped with gcc, gnu make, python, and other useful packages that allow for custom development and compilation right from the JCM. During readback or scrubbing operations, a golden copy of the configuration memory can be stored in the on-chip DRAM allowing for fast comparison and/or correction.

3.3 The Custom JTAG Carrier Card

To enable the JCM to generate and sample JTAG signals, the MicroZed is paired with a custom carrier card. This second board breaks out the signals coming from the JTAG controller state machine and routes them to a 14-pin JTAG connector. The JCM then connects directly to a target device using a JTAG cable compatible with most Xilinx FPGA boards.

An alternative to normal JTAG communication is Low-Voltage Differential Signaling (LVDS). LVDS enables the use of very long cables by mitigating signal degradation. When specified in software, the generated JTAG signals will be driven to different pins that are routed to an LVDS connector on the carrier card. When using LVDS, signals are translated back to normal JTAG before reaching the target FPGA board since few target boards support LVDS. This is done using a separate custom LVDS-to-JTAG translation board.

Finally, to enable greater functional flexibility of the JCM, the carrier card breaks out 40 additional GPIO pins. These provide an extra interface that is available to the user as needed for use in a wide variety of applications. For example, these additional pins might be used to support a SelectMAP interface, or provide a debugging interface for custom modifications of the JTAG hardware state machine.

This chapter has provided an overview of the JCM and helped motivate the usefulness of such a tool. The rest of this thesis will describe the JCM in greater detail and
discuss several of its applications. The coming chapter focuses specifically on the JTAG controller module implemented in FPGA fabric and how it pushes the boundaries of JTAG communication speed.
Chapter 4

The JTAG Hardware Controller

This chapter is dedicated to describing the JTAG hardware controller module that is implemented in the programmable logic of the Zynq 7000 SoC. The first section provides background information about why the JCM generates the JTAG signals in hardware. The second section then describes in detail the JTAG hardware controller circuit which is implemented on the Zynq FPGA. Finally, the third section discusses the mechanism used to automatically calibrate the JCM speed. The JTAG hardware controller is central to this work because it is what allows the JCM to generate JTAG signals at high speeds.

4.1 JTAG Signal Generation and Sampling

In order to communicate with the DUT, the JCM must generate and sample signals on the physical pins of the Zynq FPGA. The logic that decides exactly when these pins are driven or sampled can be implemented in software or hardware. Each of these options have significant benefits as well as drawbacks. This section discusses the two alternatives and justifies the use of a hardware circuit as the more ideal choice for the JCM.

4.1.1 The Software Approach

Using software for determining the cycle-by-cycle voltage levels of physical pins is commonly called the “bit banging” approach [19]. An example might be a micro-controller that runs software to implement a communication interface using GPIO pins. This approach requires the software to define the value on each JTAG signal at each clock cycle. A major challenge is being able to sample data that is received back from the DUT at the proper time, requiring real-time precision [20]. Without this timing precision, the processor might be busy with some other high-priority task at the moment a signal is received and cause
data to be lost. These constraints keep this approach from achieving the high speeds that are possible with a dedicated hardware circuit.

Another drawback is that this software approach is much more processor intensive than supporting a dedicated hardware circuit. Converting a large amount of data into a long sequence of high and low voltages on a physical pin is a relatively repetitive task that needlessly monopolizes processor resources that could otherwise be performing other important tasks [20]. Designating a peripheral hardware circuit to perform such tasks requires much less processor interaction.

Despite these drawbacks, a software implementation can be a better choice for some applications. The software approach is typically much cheaper and more flexible than a dedicated hardware circuit. For example, if a communication interface is subject to possible future changes, then implementing the logic for generating and sampling signals might be best in software since it is much easier to modify than a dedicated hardware circuit.

4.1.2 The Hardware Approach

An alternative to controlling the low-level signal values in software is the use of a dedicated hardware circuit. This allows the software to focus on what data is sent instead of how it is sent. Using a dedicated hardware circuit provides real-time precision and thus higher speeds. It also frees up valuable processor resources.

Compared to a software solution, a dedicated hardware circuit can be relatively expensive. Creating a custom ASIC for this purpose is only a possibility for larger scale applications. An alternative is the use of programmable logic to implement the low-level signal generation circuit. Using FPGAs not only presents a cost-effective solution for some applications but also provides a certain amount of functional flexibility that is absent when using an ASIC.

4.2 The JTAG Hardware Controller

With high speed being a top priority for the JCM, a dedicated hardware circuit for performing low-level JTAG signal generation and sampling is the better choice. For this purpose, a small hardware JTAG controller implemented in the Zynq’s programmable fabric
was developed. The precise timing necessary for high speed JTAG communication is much more easily controlled and optimized with a hardware circuit than in software. Furthermore, since the circuit is implemented in FPGA fabric, it still maintains a high level of flexibility, allowing for future changes and updates.

![JTAG Controller State Machine](image)

**Figure 4.1: JTAG Controller State Machine**

The JTAG controller consists of a small set of registers and a simple state machine. The registers are used for buffering data, receiving control signals, and reporting hardware status. The state machine coordinates the sending and receiving of JTAG signals between the JCM and the DUT. A diagram of the state machine is shown in Figure 4.1.

4.2.1 The JTAG Hardware Controller Registers

The JTAG hardware controller contains several data registers that are used to buffer the data that is shifted in and out of the DUT. Two of these registers hold the TMS and TDI values that are shifted out to the DUT during a transaction. They are pre-loaded from
software with data before the start of a transfer. A third register captures the TDO values that are shifted in from the DUT. The final TDO data can then be read by software.

For large transactions, a burst mode is available where instead of pre-loading single registers with data, the software fills a TDI FIFO. During the transaction, the data in the TDI FIFO is shifted out to the DUT. At the same time, incoming data is saved in a TDO FIFO. Once the transaction completes, the software can read out the data that was stashed in the TDO FIFO. By using FIFOs, large chunks of data can be transferred with only minimal user interaction needed. The user must simply load the TDI FIFO, initiate the transfer, and then read the TDO FIFO when the transfer has completed.

Another register, called the control register, is used to initiate transfers and specify important information about their type and length. For example, a given transfer might be characterized as a single, MSB-first, 32-bit transfer. These parameters can all be specified by modifying the bits of the control word that is sent from software to the JTAG Hardware Controller. An example software sequence of reading and writing to hardware registers is shown in Figure 4.2. In this example, the TMS and TDI registers are loaded and then a control word is created and written to the control register. This initiates the transaction, so the program waits until the JTAG hardware controller has completed its operation, and then reads out the resulting TDO values from the TDO register.

There are also bits in this register that instruct the hardware controller to hold TMS or TDI at zero for a specified length of time instead of actually providing data that contains all zeros. This is useful because TDI and TMS values are frequently ignored depending on the operation. For example TDI values are typically disregarded during a read operation while the DUT is driving data onto TDO, and TMS is always held to 0 in the middle of a data transfer to avoid prematurely ending the transaction. Time is saved by automatically generating signal values rather than consuming processor resources by passing large sequences of all zeros.

Finally, the control register is also used to specify the number of extra data shifts. Depending on clock speed and cable length these extra shifts may be necessary to compensate for timing delay. This is discussed further in Section 4.3. A full description of this register is found in Table 4.1.
// Load TDI values into the TDI register
write_tdi_reg(tdi_address, tdi_values);
// Load TMS values into the TMS register
write_tms_reg(tms_address, tms_values);
// The control word specifies the transfer type
// and initiates the transfer
int control_word = MSB_MASK | INIT_SINGLE_TRANS_MASK |
32_BIT_TRANS_MASK | CLEAR_TMS_MASK | JTAG_EN_MASK |
SINGLE_WORD_TRANS_MASK | ONE_EXTRA_SHIFT_MASK;
// Load the control word into the control register
write_control_reg(control_address, control_word);
// Wait for transfer to finish
sleep_until_done();
// Read out the TDO values from the TDO register
tdo_values = read_tdo_reg(tdo_address);

Figure 4.2: Sequence of JTAG Hardware Controller Register Access

Finally, there exists a status register which provides important information about the state of the JTAG module. The software can check this register to see information about read and write FIFOs as well as whether or not a transaction has completed. It is an important window into the operation of the module. The different signals included in this register are shown in Table 4.2.

4.2.2 The JTAG Hardware Controller State Machine

To perform a typical JTAG transaction, the application software running on the processor first loads the desired TDI and TMS data into the designated registers. The control register is then loaded with transaction settings and the initiation command, which moves the state machine from Idle to Start, thereby passing to the hardware module control over the rest of the transaction. Once the hardware controller takes over, no more interaction is required from the user while the transaction takes place. An example JTAG transaction is shown in Figure 4.3.

The Start state is used to prepare for the beginning of a transaction. In this state, the first bits of data to be transferred are shifted into I/O flip-flops. The state machine then begins the transaction by moving to the Low Clock state.
### Table 4.1: The JCM Hardware Control Register Bits

<table>
<thead>
<tr>
<th>Name</th>
<th>Bit Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Word Transfer Count</td>
<td>[0:5]</td>
<td>Specifies how many bits (1-32) of data to transfer.</td>
</tr>
<tr>
<td>Initiate Single Word Transfer</td>
<td>6</td>
<td>A value of 1 starts the transfer of a single word (32 bits or less).</td>
</tr>
<tr>
<td>Initiate Burst Transfer</td>
<td>7</td>
<td>A value of 1 starts a burst transfer which completes when the FIFO is empty.</td>
</tr>
<tr>
<td>Reset Controller</td>
<td>8</td>
<td>Resets the JTAG control hardware module</td>
</tr>
<tr>
<td>MSB First</td>
<td>9</td>
<td>A value of 1 indicates that the data should be sent MSB first, otherwise the data is sent LSB first.</td>
</tr>
<tr>
<td>Clear TMS</td>
<td>10</td>
<td>Holds the TMS value low. This is used during data transfers so that the TAP state machine does not change states.</td>
</tr>
<tr>
<td>Clear TDI</td>
<td>11</td>
<td>Holds the TDI value low. This is used during TAP state machine transitions.</td>
</tr>
<tr>
<td>JTAG Enable</td>
<td>12</td>
<td>A value of 1 allows the JTAG signals TCK, TMS, and TDI to be driven. Otherwise they are held in a high-impedance state.</td>
</tr>
<tr>
<td>Extra TDO Shift Count</td>
<td>[13:15]</td>
<td>Specifies how many extra shifts are needed at the end of a transaction to account for any timing delay.</td>
</tr>
</tbody>
</table>

**Figure 4.3: JTAG Transaction Waveform**

In the Low Clock State, the TCK signal is driven low in preparation for the first rising clock edge. Next, the state machine moves to the High Clock state, where TCK is held high. This clock edge results in the first bits of TDI and TMS being sent to the DUT. The TDO signal arriving from the DUT is also sampled on the rising clock edge, although
Table 4.2: The JCM Hardware Status Register Bits

<table>
<thead>
<tr>
<th>Name</th>
<th>Bit Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller Busy</td>
<td>0</td>
<td>A value of 1 indicates that the JTAG controller is currently performing a transaction</td>
</tr>
<tr>
<td>TDI FIFO Count</td>
<td>[1:10]</td>
<td>Indicates the current number of data words in the TDI FIFO.</td>
</tr>
<tr>
<td>TDO FIFO Count</td>
<td>[11:20]</td>
<td>Indicates the current number of data words in the TDO FIFO.</td>
</tr>
</tbody>
</table>

there may not be data available yet on the first rising edge of a transaction. This sampled TDO value is shifted into a register that buffers the data received from the DUT. As the state machine then transitions back to the Low Clock State the registers containing the TMS and TDI values to be sent are shifted to make the next bits of data available to the DUT.

This process is repeated until the specified number of bits has been transferred and the state machine transitions to the Extra Shift states. These states are used to account for the extra delay that incoming data takes to arrive. Typical transactions only require a single extra clock to get the last bit of data from TDO, but depending on speed and cable length, this number can increase. These extra states are essential to automatic speed calibration (which is discussed in Section 4.3. When this process completes, the state machine returns to the Idle State and the transaction has completed.

4.2.3 The JTAG Hardware Controller System Integration

The JTAG controller module is implemented in the FPGA fabric that is coupled with the ARM processor. The JTAG controller uses a small percentage of the resources available on the Zynq chip, as shown in Table 4.3. This leaves plenty of resources available for use by other modules should the user wish to provide additional hardware functionality.

The JTAG hardware controller is connected to the ARM processor as a hardware peripheral, allowing for easy communication between the two. Figure 4.4 shows a block diagram of the interaction between the processor and the JTAG state machine. The bus interface between the processor and hardware allows the application software to simply write data and commands to specific memory locations. The memory locations are all offsets of the
base address in physical memory of the hardware peripheral. All communication between the hardware module and the processor is done by reading and writing to these memory locations which map to registers within the JTAG Hardware Controller. As described in Section 4.2.1, the JTAG hardware controller registers include those designated for buffering TMS, TDI, and TDO data for both single and burst transactions, as well as control and status registers.

<table>
<thead>
<tr>
<th>FPGA Resource</th>
<th>% Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4%</td>
</tr>
<tr>
<td>LUTs</td>
<td>8%</td>
</tr>
<tr>
<td>Slices</td>
<td>6%</td>
</tr>
<tr>
<td>IOs</td>
<td>13%</td>
</tr>
<tr>
<td>RAMB36E1</td>
<td>3%</td>
</tr>
<tr>
<td>BUFGs</td>
<td>6%</td>
</tr>
</tbody>
</table>

Figure 4.4: JCM Block Diagram
4.3 Speed Calibration

One important aspect of the JTAG hardware controller is its ability to run at a variety of speeds. This is a desirable quality because it makes the JCM compatible with a wide variety of target devices as well as different cable lengths. The operating speed of the JCM can be automatically maximized using a calibration script, or can be set manually using a bash script. Since the system clock is used to synchronize the state machine described in Section 4.2.2, the JTAG clock actually runs at about half the speed of the system clock.

While shorter cables allow for higher speeds, longer cables may be necessary for applications where the JCM needs to be far away from the DUT. For example, in Figure 4.5 three JCMs are being used to monitor three FPGAs that are being irradiated. In this case, the JCMs need to be kept far away from the radiation beam to avoid failure. This is made possible with the use of 4 foot single-ended JTAG cables.

![Figure 4.5: Radiation Testing with Long JTAG Cables](image)

Lengthening a JTAG cable has two primary effects. First, the signal integrity decreases due to electromagnetic interference. Ways to overcome this include slowing down the clock or using Low-Voltage Differential Signaling (LVDS) which reduces signal noise over long distances. Second, using longer cables increases the delay of returning data. This can
also be solved by slowing down the clock, but an alternative method that allows for higher speeds is to delay the data capture mechanism that shifts incoming bits into a register. With these factors in mind, it is beneficial for the operating speed of the JCM to be easily adjustable.

Through a speed calibration process, the JCM determines the delay of returning data from the DUT and adjusts its data capture mechanism accordingly. This calibration begins by setting the clock to a slow speed and reading the ID code. If it is successful then the speed is increased and the process is repeated. Eventually, a speed is reached at which the ID code read is not valid because the data returns a clock cycle after it is expected. To overcome this, the data capture mechanism is delayed by one clock cycle resulting in a valid data capture again. This process of increasing speed and/or data capture delay continues until signal degradation becomes too great and the data delay is no longer the limiting factor. When this completes, the highest speed that still returned valid data is used.

Figure 4.6: Using An LVDS Cable With The JCM
The latency in clock cycles of returning data can vary depending on the clock frequency and the length of the JTAG or LVDS cable. At slow speeds or with short cables, returning data appears on the TDO signal less than one clock period after a request has finished. However, with much longer cables, this delay increases significantly. Using a 6 inch JTAG cable alone, the JCM can communicate with a Xilinx Kintex UltraScale FPGA running at up to 35 MHz with no extra shifts and 50 MHz with 1 extra shift of its TDO capture shift register. When the JTAG cable is increased to 4 feet the JCM must slow down to 25 MHz with no extra shifts and 40 MHz with 1 extra shift. Using a 15 foot LVDS cable which is then translated back to the 6 inch JTAG cable as show in Figure 4.6, the JCM can run at 12 MHz with 0 shifts, 25 MHz with 1 shift, 35 MHz with 2 shifts, and finally 45 MHz with 3 extra shifts. This example shows how without this calibration, clock frequency and cable length would be greatly limited.

The effect of this delay can be seen in Figure 4.7. The top half shows the timing of a normal JTAG transaction with the response on TDO arriving at the JCM before the next rising edge of TCK. The bottom half shows how long cables or high speeds can cause the response to arrive multiple clock cycles later. By accounting for this delay, the JCM is able to operate at high JTAG speeds despite incoming data arriving multiple clock cycles after it is normally expected. Using short cables, a max speed of 60 MHz is achieved for communication with some Xilinx 7-Series and UltraScale devices.

**Figure 4.7:** Timing Delay Fixed By Dynamic Calibration
This chapter has described in detail the JTAG controller module implemented in the FPGA fabric of the JCM. This module enables the JCM to run at high speeds, but also allows for the use of variable length JTAG cables. It acts as a hardware peripheral to the processor running the JCM software library which is the focus of the next chapter.
Chapter 5

The JCM Software Library

This chapter will describe the JCM software library. This is important to help the reader understand how the JCM makes JTAG both easier to use and highly programmable for many applications. The first section describes in detail the software architecture, and the second section discusses the Python scripting interface.

The JCM software library is designed to simplify the creation of custom JTAG applications. It provides an abstraction between the low-level details of the JTAG state machine and the high-level FPGA operations such as those needed for reliability applications. This abstraction is important because the complexity of a typical JTAG transaction would make it very difficult to create each one from scratch. For example, a typical JTAG transaction generated by the JCM has to navigate the JTAG state machine, vary the bit-endianness of data depending on the target, assemble valid Xilinx configuration packets (see Appendix A), and handle the overhead of multi-device JTAG chains. The high-level interface that this software library provides makes it very easy to create new custom JTAG sequences without needing to understand such underlying details.

The embedded ARM processor runs Linux to simplify the user interface and to allow users of the JCM to develop, debug, and compile applications directly on the device. Not only can C/C++ executables be created, but a Python interface to the underlying C/C++ library makes test scripting also available. This flexibility and ease of modification makes the JCM a useful tool for a wide variety of testing applications.

5.1 Software Architecture

The JCM software architecture is layered to provide several levels of abstraction as shown in Figure 5.1. The lower levels, which include the Firmware and JTAG levels, provide
maximum flexibility and control. However, using them directly requires an understanding of the JTAG protocol and often the device specific register interface as well.

The higher levels of the software architecture, which include the Device and User API levels, offer a simplified interface for accessing the FPGA configuration memory (CRAM). They handle device differences as well as provide many common applications that are useful for CRAM access. Although the higher levels include many common applications, the user may wish to develop using the lower levels when implementing new features or supporting new target device vendors. The following subsections describe each of these levels in detail.

![Diagram of JCM Software Layers of Abstraction]

**Figure 5.1:** The JCM Software Layers of Abstraction

### 5.1.1 Firmware Level

The firmware level of the JCM software library is the lowest and most customizable interface to the JTAG hardware controller. It provides an abstraction to the implementa-
tion details of the hardware module design. This level provides no abstraction, and allows the user to control every bit that is sent over the JTAG cable by the hardware controller. Communication with the hardware is done by memory mapping the hardware space to user space. This way, the software can read and write data to the JTAG hardware controller by writing to these shared memory locations. This level of control can be useful when debugging the JTAG hardware controller or when testing new JTAG sequences. This layer can be the most difficult to use, however, since all the complexities of JTAG transactions must be taken into consideration. The higher layers of the software architecture provide these abstractions making the JCM much easier to use.

At this level, there are two primary ways to perform JTAG transactions: single-word and burst. The former entails reading or writing 32 bits of data or less, while the latter involves the use of a FIFO for multi-word transfers. To perform a JTAG transaction, data is loaded into either a register or a FIFO and then a control register is modified to trigger the shifting out or in of data with a state machine. For example, a transaction to reset the JTAG state machine would be done by calling the `changeState()` command with the value `0b11111` and a transfer length of 5 as the parameters. This function first loads the specified value into the hardware TMS register with the `writeTmsReg()` function. It then creates a control word that contains the transfer length, the bit-endianness, and a bit signifying that only a single transfer is needed. When this control word is written to the hardware control register with the `writeControlReg()` function, the JTAG transaction is initiated by the hardware. The full example sequence can be seen in Figure 5.2, and a full list of functions available in the Firmware-level API is found in Table 5.1.

5.1.2 JTAG Level

The next level of abstraction in the JCM software library is the JTAG level. This level hides the complexities of the JTAG TAP state machine (shown in Figure 2.1) and provides a set of commands that read and write to the JTAG data and instruction registers. These commands also handle the extra overhead required when there are multiple devices on the JTAG chain. This abstraction makes it easier for users without in-depth knowledge of the JTAG state machine and daisy-chaining to create custom JTAG sequences. For example,


```c
// Set value to be written to the TMS buffer
int tms_val = 0b11111; // Resets the JTAG state machine
// Set number of bits to send over JTAG
int num_bits = 5;
// Call function to change the JTAG TAP state
changeState(tms_val, num_bits);

// Change State function definition
void changeState(u32 command, int lengthInBits){
    // Load the TMS buffer with the reset command
    perform_single_write(TMS_ADDRESS, tms_val);
    // Create a control word
    control_word = INIT_TRANSFER_MASK | num_bits;
    // Write to the control register to initiate the JTAG transfer
    write_control_register(control_word);
}
```

**Figure 5.2:** Resetting the JTAG State Machine With Firmware-Level Commands

when there are multiple devices on the JTAG chain, the user must typically bypass all of
the devices except the one under test. This requires the user to provide longer IR values as
well as ignore the extra bits that accompany the data when reading or writing the DR.

A typical JTAG register query is shown in Figure 5.3. It involves first specifying the
register to be read, and then performing the read operation itself. To specify the target reg-
ister, the JCM command `resetToShiftIr()` is called which moves the JTAG state machine
from the RESET state to the SHIFT IR state. The target register ID is then passed as a param-
eter to the `writeToIr()` command which specifies which JTAG destination register is the tar-
get. To read the target JTAG destination register, the JCM command `exitIrToShiftDr()`
moves the JTAG state machine from the EXIT IR state to the SHIFT DR state. Finally, a call
to the `readFromDr()` command reads out the desired register value. A full list of JTAG-level
commands can be found in Table 5.2.

### 5.1.3 Device Level

The third level of the JCM software library is the Device Level. This level provides a
set of commands that perform basic functions that are specific to the type of devices being
### Table 5.1: Firmware-Level Library Functions

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void <code>changeState</code>(u32 command, int lengthInBits)</td>
<td>Changes the JTAG State Machine state.</td>
</tr>
<tr>
<td>void <code>performSingleTransfer</code>(u32 tmsValues, u32 tdiValues, int lengthInBits, bool readTdo, u32 endianMask)</td>
<td>Performs a single JTAG transaction with the specified TDI and TMS values. If readTdo is true then return the resulting TDO data.</td>
</tr>
<tr>
<td>void <code>performBurstTransfer</code>(u32 *tdiValues, int lengthInWords, bool readTdo, u32 endianMask)</td>
<td>Performs a burst JTAG transaction using the given array of TDI values.</td>
</tr>
<tr>
<td>u32 <code>readHardwareVersion</code>()</td>
<td>Reads the version number of the hardware design.</td>
</tr>
<tr>
<td>void <code>disableJtag</code>()</td>
<td>Puts the JTAG output pins into a high-Z state.</td>
</tr>
<tr>
<td>bool <code>hardwareBusy</code>()</td>
<td>Checks the hardware status to see if a transaction is currently being executed.</td>
</tr>
<tr>
<td>void <code>setExtraShiftMask</code>(int numExtraShifts)</td>
<td>Tells the JTAG hardware controller to delay its data capture mechanism by a given number of extra clock cycles.</td>
</tr>
<tr>
<td>void <code>writeTdiReg</code>(u32 baseAddress, u32 tdiValues)</td>
<td>Writes the specified data to the hardware TDI register.</td>
</tr>
<tr>
<td>void <code>writeTmsReg</code>(u32 baseAddress, u32 tmsValues)</td>
<td>Writes the specified data to the hardware TMS register.</td>
</tr>
<tr>
<td>void <code>writeControlReg</code>(u32 baseAddress, u32 controlWord)</td>
<td>Writes the specified data to the hardware control register.</td>
</tr>
<tr>
<td>void <code>writeTdiBurstReg</code>(u32 baseAddress, u32 tdiValues)</td>
<td>Writes the specified data to the hardware TDI burst register, which is the input to a FIFO.</td>
</tr>
<tr>
<td>u32 <code>readTdoSingleReg</code>(u32 baseAddress)</td>
<td>Reads the value of the hardware TDO register.</td>
</tr>
<tr>
<td>u32 <code>readTdoBurstReg</code>(u32 baseAddress)</td>
<td>Reads the value of the hardware TDO burst register, which is the output of a FIFO.</td>
</tr>
<tr>
<td>u32 <code>readStatusReg</code>(u32 baseAddress)</td>
<td>Reads the value of the hardware status register which indicates if a JTAG transaction is currently occurring.</td>
</tr>
</tbody>
</table>

### Table 5.2: JTAG-Level Library Functions

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void <code>resetStateMachine</code>()</td>
<td>Moves the JTAG TAP state machine to the Reset state.</td>
</tr>
<tr>
<td>void <code>resetToShiftDr</code>()</td>
<td>Moves the JTAG TAP state machine from the Reset state to the Shift-DR state.</td>
</tr>
<tr>
<td>void <code>resetToShiftIr</code>()</td>
<td>Moves the JTAG TAP state machine from the Reset state to the Shift-IR state.</td>
</tr>
<tr>
<td>void <code>exitDrToShiftIr</code>()</td>
<td>Moves the JTAG TAP state machine from the Exit-DR state to the Shift-IR state.</td>
</tr>
<tr>
<td>void <code>exitIrToShiftDr</code>()</td>
<td>Moves the JTAG TAP state machine from the Exit-IR state to the Shift-DR state.</td>
</tr>
<tr>
<td>void <code>exitToIdle</code>()</td>
<td>Moves the JTAG TAP state machine from the Exit-DR or Exit-IR state to the Idle state.</td>
</tr>
<tr>
<td>void <code>writeToIr</code>(u32 registerVal)</td>
<td>Writes the specified value to the JTAG Instruction Register (IR).</td>
</tr>
<tr>
<td>void <code>writeToDr</code>(u32 *dataIn, int dataLength)</td>
<td>Writes the given data array to the JTAG Data Register (DR).</td>
</tr>
<tr>
<td>u32 * <code>readFromDr</code>(int dataLength)</td>
<td>Reads the specified length of data from the JTAG Data Register (DR).</td>
</tr>
</tbody>
</table>
// Move TAP controller to the SHIFT_IR
reset_to_shift_ir();
// Write the register id to the IR
write_jtag_instruction_register(reg_id);
// Move TAP controller to the SHIFT_DR
shift_ir_to_shift_dr();
// Read the register value from the DR
int reg_value = read_jtag_data_register();

**Figure 5.3:** Reading a Register with JTAG-level Commands

targeted. These commands take advantage of configuration files that are read in and specify the details of the target device such as the ID code, the JTAG chain size, and the device type. With this information, the commands at the device level customize each function to the target device, allowing the user to simply swap configuration files to switch between devices and have all commands at this level still work.

The JCM has been generally used for FPGA applications, so this level has been written to handle operations such as configuring, reading and writing registers, and accessing the CRAM. These commands abstract away the need to assemble packets of command sequences in order to communicate with the Xilinx Configuration Registers. Configuration files for Xilinx FPGAs also include the device family, the part name, the path to a default programming file, and the path to a file including all of the valid configuration frame addresses. Supported devices include Virtex-5, Virtex-6, and all 7-Series, UltraScale, and UltraScale+ devices. An example configuration file is found in Figure 5.4.

A common sequence of commands at this level might include configuring the target FPGA and then checking to make sure it configured properly. To do this, the user would call the `configure_fpga` command with the name of the desired bit file. Next the `read_config_register` command is called with the ID of the status register as a parameter. The return value of this register can be used to determine whether configuration was successful. This full example is shown in Figure 5.5 and a full list of Device-level commands can be found in Table 5.3.
Figure 5.4: A JCM Configuration File For A Xilinx Kintex-7 FPGA

```c
// Configure the FPGA with the default .bit file
configureDevice();
// Read the Status Register
u32 status = readConfigRegister(STATUS_REGISTER_CODE);
// Verify that configuration was successful
// by checking the Done bit
if(status & DONE_MASK){
    // Configuration was successful
} else {
    // Configuration failed
}
```

Figure 5.5: Example Configuration Sequence Using Device-Level Commands

5.1.4 User API Level

The top level of the JCM Software Library is the User API. Like the Device Level, this level is also geared toward specific device types. This section describes the functions
that are used specifically for Xilinx FPGAs. This level builds upon the generic device-
level commands by providing a wide variety of specific commands that perform common
operations. While the device-level functions often require complex parameters, the functions
at this level often require no parameters at all. For example, the device level has a generic
readConfigRegister() command that requires a register-read command as a parameter.
This read command is a somewhat complex sequence of bits that can be assembled with
help from Xilinx documentation. In contrast, the User API level has a series of methods
that each read a specific register and contain the read-command constants embedded within
them. This level also has various functions that perform common sequences of the simple
device-level commands. These are all provided to facilitate the use of the JCM.

These high level sequences of commands also serve as templates for custom develop-
ment. Since some functions at this level perform relatively specific sequences of operations,
the user may want to create a custom function with slightly different behavior. To accom-
plish this, the user can follow the various example functions provided at this level to develop
new ones.

A typical sequence of commands operating at the User API Level might entail reading
the device ID code, calibrating the JTAG speed, and injecting a fault. Reading the ID code
verifies that communication with the device is good. JTAG calibration finds the maximum
JTAG speed that the target device supports and the necessary extra data capture shifts
associated with it (see Section 4.3). Injecting a fault can be used to simulate the effects of

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void writeConfigRegister(u32 writeCommand, u32 regValue)</td>
<td>Writes the given value to the Xilinx Configuration Register specified by the write-command.</td>
</tr>
<tr>
<td>u32 readConfigRegister(u32 readCommand)</td>
<td>Reads the Xilinx Configuration Register specified by the read-command.</td>
</tr>
<tr>
<td>void writeConfigFrame(u32 beginFrameAddr, int numFrames, u32 * dataIn)</td>
<td>Writes the given array of data to the Xilinx Configuration Memory beginning at the specified address.</td>
</tr>
<tr>
<td>u32 * readConfigFrame(u32 beginFrameAddr, int numFrames)</td>
<td>Reads the specified number of frames from the Xilinx Configuration memory beginning at the given address.</td>
</tr>
<tr>
<td>void configureDevice()</td>
<td>Configures the FPGA with the default bit file specified in the JCM configuration file.</td>
</tr>
</tbody>
</table>
ionizing radiation. All of these functions can be performed with little in-depth understanding of JTAG or the Xilinx Configuration Registers, and are run with few if any parameters. This sequence is illustrated in Figure 5.6, and a full list of User API functions can be found in Table ???. Some of the functions included in the table are explained further in the next chapter.

```c
// Read the device ID code to see if communication is good
int id_code = read_id_code();
// Perform automatic speed calibration (0 indicates max speed)
bool success = calibrate_device(0);
// Inject a 3-bit fault at a specified location in CRAM
int frameAddress = 0;
int word = 100;
int bit = 20;
int numBits = 3;
bool success = injectFault(frameAddress, word, bit, numBits);
```

**Figure 5.6:** Commands of a Typical JCM Session

The User API level is the top level of the JCM Software Library, but the creation of actual Linux executables requires a top-level main function that parses the command line and calls JCM library functions. The JCM Linux image contains a set of top-level examples. These example top level files are generally no more than a wrapper around the JCM library functions, but they are likely where most custom user-developed code would reside.

### 5.2 Scripting Interface

Despite containing many common example applications such as fault injection and scrubbing, the JCM is also designed for custom test development. As mentioned above, this can be done by modifying/adding to the JCM Software Library, or by creating custom top-level files containing user-developed code. A common alternative to modifying underlying C/C++ libraries is compiling the libraries into a shared library object that is available to scripting languages.
### Table 5.4: User API Library Functions

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32 readIdCode()</td>
<td>Reads the Xilinx Configuration ID Code Register.</td>
</tr>
<tr>
<td>u32 readFar()</td>
<td>Reads the Xilinx Configuration Frame Address Register (FAR).</td>
</tr>
<tr>
<td>u32 readStatus()</td>
<td>Reads the Xilinx Configuration Status Register.</td>
</tr>
<tr>
<td>void writeFar(u32 value)</td>
<td>Writes the given value to the Xilinx Configuration Frame Address Register (FAR).</td>
</tr>
<tr>
<td>void fullDeviceConfigure()</td>
<td>Configures the FPGA with the default bit file specified in the JCM configuration file.</td>
</tr>
<tr>
<td>u32 * readFullDevice(string fileName, bool readBram)</td>
<td>Reads the entire configuration memory and saves it to a file. The inclusion of BRAM data is optional.</td>
</tr>
<tr>
<td>void setActiveDeviceIndex(u32 index)</td>
<td>Sets which device on the JTAG chain the JCM interacts with.</td>
</tr>
<tr>
<td>void calibrateDevice(int targetSpeed)</td>
<td>Sets the speed of the JTAG hardware controller. The speed can either be specified or it defaults to the maximum that is supported by the target device.</td>
</tr>
<tr>
<td>bool injectFault(u32 frameAddress, int wordNum, int bitNum, int numBits)</td>
<td>Injects a fault into the specified frame, word, and bit. Injections can be multi-bit.</td>
</tr>
<tr>
<td>void blindScrub(bool initReadback)</td>
<td>Continuously writes the golden data into the CRAM. An initial readback or a saved file can be used as the golden data.</td>
</tr>
<tr>
<td>void readbackScrub(bool initReadback, bool readBram, bool fixErrors, bool injectFaults)</td>
<td>Repeatedly reads the whole CRAM (BRAM optional) and compares it against the golden data. Faults can be injected in between each readback iteration.</td>
</tr>
<tr>
<td>void hybridScrub(bool initReadback)</td>
<td>Monitors the internal scrubber (Readback CRC). When uncorrectable errors occur, the JCM steps in and corrects them.</td>
</tr>
</tbody>
</table>

To facilitate the use of the JCM Software Library, the JCM features a Python interface to the underlying C/C++ library. Creating a custom test script begins with compiling and importing the JCM Python module into a new script. Then, a handle to an object encapsulating the JCM Software Library can be obtained from an included configuration file. Once this object is returned, the user can create a custom script by calling different combinations of the available methods and/or custom functions.

This chapter has summarized the architecture of the JCM Software Library. Each level of the architecture provides a different level of abstraction and complexity, allowing the user to interface with the JCM in a variety of different ways. The leveled structure also allows for greater modularity and re-usability. This flexibility has allowed the JCM to be used for a wide variety of applications which will be discussed further in the next chapter.
Chapter 6

JCM Applications

The JCM hardware and software are used together to create a variety of useful applications for JTAG based systems. This chapter will provide an in-depth discussion of how the JCM is used for radiation testing, which is a common scenario where the JCM is a very effective solution. This chapter also discusses a number of other JCM applications including FPGA configuration, fault injection, processor debugging, temperature and voltage monitoring, and the debugging and monitoring of FPGA designs.

6.1 FPGA Configuration

Full configuration is the most elementary FPGA function of the JCM, and one where high speed is very important. This operation consists of streaming in the data contained in a programming (.bit) file. These files are generated by FPGA vendor CAD tools and contain the information to program the FPGA’s programmable logic and block RAM.

FPGA configuration is typically performed over JTAG since nearly all FPGA boards contain a JTAG port of some kind. FPGA vendors provide software and hardware tools to facilitate this operation; however, these tools are typically not designed for high-speed or programmability. With modern devices growing increasingly large, FPGA configuration can take a significant amount of time using a serial connection to program bit streams that are tens of MB in size. At maximum speed (around 60 MHz), the JCM can configure a Xilinx Kintex 7 (xc7k325t) device in 2.3 seconds, while the Xilinx iMPACT tool takes a full 12 seconds. That 5X speedup can make a significant difference when frequent configurations are needed.
6.2 Configuration Scrubbing

The JCM can combine various basic functions to form extremely complex and powerful applications. One of the most common uses of the JCM is FPGA configuration scrubbing which involves monitoring the FPGA’s configuration memory (CRAM) state and correcting errors that may occur due to ionizing radiation. This section describes three different configuration scrubbing mechanisms implemented by the JCM, while the following section discusses how they are used for radiation testing. The various types of scrubbers implemented by the JCM illustrate its flexibility and programmability.

Configuration scrubbing entails detecting and dynamically correcting errors caused by radiation exposure in an FPGA’s CRAM [14][21][22]. Error correction is generally performed by either partial or full reconfiguration while the design continues to run, and needs to be fast in order to prevent build-up of errors in the design. The error correction functionality is essential for any non-radiation-hardened SRAM based FPGA that is subject to radiation upsets. Coupled with other error-mitigation techniques such as Triple-Modular Redundancy (TMR) with voting, configuration scrubbing can greatly increase the reliability and mean time to failure (MTTF) of an FPGA design [23].

Beyond its use as a mechanism for increasing reliability, configuration scrubbing is very useful for characterizing the frequency and nature of upsets caused by radiation exposure [24]. As upsets occur, a scrubber can detect and record their type and frequency for later analysis. This analysis is important for characterizing the sensitivity of a given device or design, as well as evaluating the effectiveness of the scrubber itself.

The JCM features three main types of scrubbing. Each is characterized by a variety of important factors. These include whether all errors can be detected and corrected, the average time it takes to correct an upset, how much external involvement is required, and what mechanism is used for correction. The three types of scrubbing are Blind, Readback, and Hybrid. Hybrid scrubbing utilizes a hardware scrubber that is built into modern Xilinx FPGAs that will also be discussed in this section.
6.2.1 Blind Scrubbing

The first type, called blind scrubbing, repeatedly overwrites the FPGA CRAM from a golden copy [25]. This golden copy is obtained through an initial readback, and is stored in on-chip DRAM. This straightforward method of scrubbing performs no readback after creating the initial golden copy and thus lacks any means of error detection. Despite this fact, this scrubbing method is completely robust in its ability to correct all upset types at a constant speed. Figure 6.1 shows an example of JCM code implementing blind scrubbing.

```c
// Perform an initial readback to obtain a golden
// copy of the CRAM
u32 * goldenData = readFullDevice();
// Specify starting frame address
u32 startAddr = 0;
// Continuously write the golden data to the CRAM
while(1){
    writeConfigFrames(startAddr, NUM_DEVICE_FRAMES, goldenData);
}
```

**Figure 6.1:** Blind Scrubbing Using the JCM

6.2.2 Readback Scrubbing

The second type, called readback scrubbing, continuously reads the full CRAM instead of overwriting it [26][25]. After readback is finished, the data is compared to a golden copy of the CRAM that was previously saved in on-chip DRAM. When differences between the two are found, they are reported and then optionally corrected. This correction can be targeted to only those frames with errors in them instead of performing full-device configuration as with blind scrubbing. Although it is slightly slower than blind scrubbing, it performs error detection as well as fully robust correction. Like blind scrubbing, it also requires a golden copy of the CRAM to be stored so that errors can be corrected. Figure 6.2 shows example JCM code implementing readback scrubbing.
// Perform an initial readback to obtain a golden copy of the CRAM
u32 * goldenData = readFullDevice();
// Continuously scrub the FPGA
while(1){
    // Read the full device
    u32 * testData = readFullDevice();
    // Compare the test data to the golden data
    u32 * listOfErrorFrames = compareData(goldenData, testData);
    // Print Errors
    printErrors(listOfErrorFrames);
    // Correct the Errors
    correctErrors(listOfErrorFrames);
}

**Figure 6.2:** Readback Scrubbing Using the JCM

**Table 6.1:** JCM Function Delays on a Kintex 7 FPGA

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time to Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full FPGA Configuration</td>
<td>2.3 s</td>
</tr>
<tr>
<td>Single Configuration Frame Read</td>
<td>850 us</td>
</tr>
<tr>
<td>Configuration Register Read</td>
<td>230 us</td>
</tr>
</tbody>
</table>

Figure 6.3 shows the output of a readback scrubber monitoring an FPGA while it is being exposed to ionizing radiation. In this example, each full readback finds two different bits that were upset by radiation strikes. After reading the individual memory frames that were identified as bad a second time to confirm the presence of an error, the scrubber then reports the upset locations and corrects the faulty bits.

In the above example only two faults were found each readback iteration, but this number can vary greatly depending on the energy and flux of the radiation beam. For this reason, the readback scrubber needs to operate as fast as possible to minimize the accumulation of configuration upsets. Since readback scrubbing reads the entire configuration memory, it takes about the same amount of time as a full reconfiguration. For a Kintex 7 device (XC7k325T) readback takes approximately 2.3 seconds. Execution latency of several common operations performed on this device are shown in Table 6.1.
6.2.3 Hybrid Scrubbing

The third type, called hybrid scrubbing, is a form of readback scrubbing that takes advantage of a built-in hardware mechanism called the Readback CRC [21][26]. This module, which can be turned on for any design, performs continuous readback and uses Error Correction Code (ECC) checking to detect errors. Each CRAM frame dedicates one word for storing the ECC value that is calculated based on the data that is stored in the CRAM frame. Every time the RBCRC scan reads a frame, the ECC value is recalculated and compared against the value that was previously stored in the frame. When a mismatch occurs the RBCRC knows that a bit has been changed.

The mismatched ECC values can be used to achieve Single Error Correction, Double Error Detection (SECDED) capability. This means the RBCRC can successfully correct any single-bit error, and detects most multi-bit errors. Besides the single-frame ECC checking,
there is a device-wide CRC calculation which detects when any bit in the device is modified. Any errors that are missed by the RBCRC are caught by the whole-device CRC calculation.

The RBCRC can be set up to notify the JCM when errors are found. Most importantly, when a multi-bit error occurs, the internal scrubber is not able to fix it so the JCM steps in and corrects it using its golden copy of the CRAM. To enable communication between the RBCRC and the JCM, information about each error detected by the RBCRC is saved in a FIFO. The output of the FIFI is then connected to a BSCAN module which makes data accessible through JTAG. By monitoring this FIFI, the JCM can report any errors that are found and correct errors when needed.

This hybrid method combines the speed of built-in hardware with the robustness of normal readback scrubbing to form the fastest method of the three scrubbing types. Unlike the others, however, it does require minor target hardware design additions. These include the RBCRC itself as well as the FIFO and BSCAN modules. An example JCM program that performs hybrid scrubbing is shown in Figure 6.4.

```
// Perform an initial readback to obtain a golden
// copy of the CRAM
u32 * goldenData = readFullDevice();
// Continuously monitor the FPGA
while(1){
    u32 * bscanOut = readBscan();
    // Parse the output of the BSCAN module
    u32 errorAddress = bscanOut[0];
    u32 singleErrorFlag = bscanOut[1];
    // Print Errors
    printError(errorAddress);
    // Correct the error if it was a multi-bit error
    if(!singleErrorFlag){
        correctErrors(listOfErrorFrames);
    }
}
```

**Figure 6.4:** Hybrid Scrubbing Example Using the JCM
6.2.4 Radiation Testing with the JCM

Scrubbing is especially useful during ground-based radiation testing. Radiation testing entails exposing an FPGA to high-levels of ionizing radiation to simulate the environment that an FPGA might experience when used in aerospace applications [6]. During a radiation test, the JCM uses scrubbing to detect and correct any errors that occur, and outputs logs that can be carefully analyzed later.

These tests can serve a variety of purposes, but the primary purpose of the JCM during radiation testing is to gather data on the upset rate and cross section of FPGAs. These metrics vary widely between different device technologies and vendors. This information can be used to estimate the reliability of using FPGAs in radiation environments.

Figure 6.5 shows an example of the JCM being used during a radiation test. The JCM shown is monitoring and scrubbing a Kintex 7 board as it communicates over several lanes of a Multi-Gigabit Transceiver (MGT) with another Kintex 7 board not directly in the radiation beam. The JCM is used to scrub the CRAM of the DUT as well as monitor its temperature and voltage levels (see Section 6.6).

The JCM itself is also sensitive to radiation exposure. An especially sensitive component of the JCM is the Micro-SD card that holds the Linux image and the Zynq’s programming bit stream. Early radiation tests using the JCM were hindered by failures of this and other sensitive components. Part of the motivation for adding the ability to support long JTAG cables was to allow the JCM to be kept far enough away from the radiation beam to avoid being upset itself.

A radiation test environment is where the programmability of the JCM becomes vitally important. The results of a radiation test are often very different from what is expected, resulting in the need to modify and adapt test sequences and scripts while at the testing facility. For example, during one radiation test, there were more upsets than expected, making it difficult to collect real-time statistics without manually counting each line of the console output. With a few quick changes to the software, the test started printing a count of the errors found so counting was no longer necessary. Since all JCM applications are so easily modified and recompiled, on-the-fly test adaptations are fast and simple.
High speed is also especially important in a radiation environment due to the accelerated rate at which upsets occur in a radiation test compared to what is experienced in most aerospace environments. This accelerated rate is important so that radiation tests can be performed in a few days, but simulate the effects of radiation experienced in real aerospace environments over much longer periods of time. However, this acceleration requires scrubbing that is fast enough to keep upsets from accumulating.

6.3 Fault Injection

Fault injection is another common application of the JCM. It involves manually flipping one or more bits in an FPGA’s configuration memory and then monitoring the design for failure. Since FPGA circuit behavior is determined by the values saved in CRAM, changing a single bit can disrupt the proper functioning of the circuit or put it in a bad state. Depending on the design, this bad state may not be obvious until the error has had time to
propagate. If an injected fault causes a failure, then the FPGA is typically reconfigured to put it back into a good state for future fault injection. This fault injection flow is shown in Figure 6.6.

Fault injection is used to simulate the effects of ionizing radiation. It is frequently used in the characterization of a given design’s relative sensitivity to failure [27]. Design sensitivity testing consists of repeatedly injecting faults to determine what percentage of upsets in the CRAM cause design failure. If a large number of random bits are tested in this way, a good estimate of design sensitivity can be obtained. The means of detecting a circuit failure is highly design-dependent, but a common method when using the JCM is to sample internal signals that have been exposed to JTAG using a BSCAN primitive (see Section 6.5).

The mechanism for performing fault injection on a Xilinx FPGA using the JCM is more complex than simply flipping a CRAM bit. It instead requires reading out an entire frame of configuration data, flipping one or more bits of the saved array, and then writing it back into memory. A full frame of data must be read and written because a frame is the
smallest addressable unit of CRAM (see Appendix A). Figure 6.7 shows how single bit fault injection is implemented by the JCM.

```c
// Choose random frame address index
int frameIndex = rand() % NUM_FRAMES;
// Get address from list of valid frame addresses
u32 frameAddr = validFrameAddresses[frameIndex];
// Get random word number
u32 word = rand() % NUM_WORDS_PER_FRAME;
// Get random bit number
u32 bit = rand() % 32;
// Read the chosen frame from the CRAM
u32 * frameData = readConfigFrame(frameAddr);
// Inject a fault into the frame data
frameData[word] = frameData[word] ^ (1 << bit);
// Write the frame data back into the CRAM
writeConfigFrame(frameAddr, frameData);
```

**Figure 6.7:** Random Fault Injection Using the JCM

One of the most common fault injection campaigns is sequential fault injection. It systematically tests each bit of the CRAM individually. After each bit is flipped, the design is checked for functional errors. If a configuration upset does not disrupt the design then the upset is reversed and the testing continues. If a functional error does occur then the FPGA is reconfigured. This determines precisely which bits are sensitive, and the exact design sensitivity. Unfortunately this process can take from days to weeks to complete depending on how long each error is allowed to propagate and how long device reconfiguration takes after a design failure is detected.

Another common fault injection campaign is random fault injection. In contrast to sequential, a random campaign injects faults at completely random locations throughout the FPGA. This method more accurately simulates the effects of radiation exposure. Random fault injection is typically not used to record exactly which bits are sensitive, but it can make a good estimate of overall design sensitivity within minutes. This design sensitivity is measured in the percentage of injected faults that cause a design to fail.
Design sensitivity obtained through fault injection can be used in a variety of ways. One such purpose is for analyzing the effectiveness of upset masking techniques such as Triple-Modular Redundancy (TMR). For example, fault injection was used to test both a non-TMR and a TMR version of a Leon3 soft processor design programmed into a Kintex-7 FPGA. Using random fault injection the non-TMR version was found to have a 0.37% sensitivity while the TMR version had a 0.0073% sensitivity. Figure 6.8 shows these results with sensitive frames colored red and insensitive frames colored green.

![Figure 6.8: Fault Injection Results of a non-TMR and a TMR Leon3 Design](image)

These results were collected in a relatively short amount of time allowing for quick analysis of TMR effectiveness. This effectiveness depends upon triplication coverage, the avoidance of single-point failures, and the level of granularity of TMR zones. These variables can be tweaked and then quickly re-analyzed for effectiveness.
Another purpose is to determine whether a design is sufficiently reliable to be deployed in a radiated environment. In both cases, fault injection is used as an estimate and its results must be validated by ground-based radiation testing; however, by correlating fault injection results with radiation test results, very good estimates can be obtained [28].

![The JCM Fault Injection GUI](image)

**Figure 6.9:** The JCM Fault Injection GUI

In order to help the user better conceptualize fault injection results, the JCM includes a graphical user interface for random fault injection. The GUI is written in Python and uses the Python interface to the JCM Software Library. Since it is a native GUI running on the JCM, an SSH connection with X-forwarding allows the user to interact with it. Figure
6.9 displays the fault injection GUI on the left, with a screenshot of the design under test in Vivado on the right. Each colored square on the GUI represents a single configuration frame, with red being sensitive, green being insensitive, and black being untestable. When the user selects a frame, a bit-level representation of that frame can also be seen, as shown in the upper right corner of Figure 6.9. The design shown in the figure consists of 8 PicoBlaze processors. Without using any circuit redundancy techniques, this design has an overall sensitivity of 1.3%, or in other words, 1.3% of bits tested caused the design to fail.

6.4 Processor Debugging

Another application of the JCM is processor debugging. The JCM can monitor and debug a MicroBlaze processor implemented on an FPGA [29]. It can download binary executable files, single step through programs, print out source code context, and access registers. These are many of the same functions that are available through other industry tools, but the JCM allows these applications to be scripted and combined with any other JCM functionality. This can be very useful during radiation testing, since radiation particle strikes easily upset soft and hard-core processors. Communicating directly with a processor allows for a better understanding of what specifically was affected by an upset. A screen capture of the processor debugging application menu is found in Figure 6.10.

An especially useful combination of features is processor fault injection and debugging using the same JTAG chain. When an error is detected during fault injection, the JCM can immediately be used to determine what caused the processor to fail and why. This fault-injection-debugging flow enables users to discover what parts of a processor are most vulnerable and understand the processor’s most common failure modes. For example, this
fault injection and debugging functionality has been used with NASA Goddard’s SpaceCube v2.0 [30].

6.5 Design Monitoring and Debugging

Besides reliability applications, the JCM can also be useful for hardware design status monitoring and debugging. A special Xilinx hardware primitive called a BSCAN allows the user to expose internal circuit signals to JTAG. The user must instantiate the BSCAN module in their hardware design and then attach design signals to the inputs of the BSCAN module. While the design is operating the JCM can access the signals attached to the BSCAN module by reading from one of the four USER registers supported as JTAG destination registers by Xilinx.

This mechanism is commonly used as the means of determining design failure during fault injection. When fault injecting a soft-processor design, the processor’s state or output may become corrupt. Signals that monitor the processor’s health can be attached to bits of the BSCAN primitive and read using the JCM.

BSCAN monitoring is also very useful during hybrid configuration scrubbing. This scrubbing architecture allows the internal ReadBackCRC mechanism to use ECC calculations to correct single bit errors and detect multi-bit errors. When a multi-bit error is detected, information about the upset is made available to JTAG through the BSCAN module so that the JCM can then manually scrub the error.

6.6 Temperature and Voltage Monitoring

Another module that is accessible through JTAG is the Xilinx Analog to Digital Converter (XADC), which was renamed the SYSMON for UltraScale and UltraScale+ architectures. By polling registers in the XADC, the JCM can monitor on-chip temperature and voltage levels. This can be very useful information during radiation tests since design sensitivity can depend on die temperature [31]. Figure 6.11 shows the information printed when the JCM reads from the XADC.
6.7 Remote Tool Control

The JCM can also be used to run Xilinx software tools remotely. Instead of connecting an FPGA to a host computer for communication with Xilinx tools, the Xilinx Virtual Cable can be used instead. This virtual cable is a program that translates commands from the Xilinx tools into Ethernet packets [32]. These packets can be sent to the JCM and then translated back into JTAG commands for the target device. This allows Xilinx software tools to be used from a remote location as long as connection to the JCM is possible. Figure 6.12 shows a block diagram of how the JCM allows for remote tool use.

![Figure 6.12: Using the JCM to Run Xilinx Software Tools Remotely](image)

This chapter has described several important applications of the JCM; however, there are a wide variety of other applications for which the JCM would be well suited. The layered architecture of the JCM software library allows for the easy customization of the applications discussed in this chapter, as well as the creation of new JTAG applications. The scope of this thesis is limited to using the JCM for interaction with FPGAs, but any application that
would benefit from high-speed, programmable JTAG access would be a good candidate for JCM use.
Chapter 7

Conclusion

This work introduced a tool called the JTAG Configuration Manager that provides an extensive software API for creating custom applications that run at high JTAG speeds. JTAG is used as the interface of choice due to its low overhead, universality, and scalability. The success of the JCM has shown that JTAG is a very good interface for both simple FPGA configuration as well as more complex FPGA reliability applications. This is true especially when its speed is accelerated through custom hardware and its use is facilitated by a flexible software library.

The JCM has been used for a wide variety of purposes since it was first developed at Brigham Young University. The original motivation for its creation was the need for a fast configuration scrubber to be used during radiation beam testing by researchers in the BYU Configurable Computing Laboratory. It soon became the go-to tool for all FPGA configuration interaction, including device readback, fault injection, processor debugging, and design monitoring. Several industry and government sponsors including NASA Goddard, Los Alamos National Laboratories, Sandia National Laboratories, Boeing, Ratheon, and Cisco have used the JCM for similar purposes.

Since the JCM is easily adapted to work with any Xilinx FPGA, further work will include ensuring compatibility with future generations of Xilinx FPGAs. The JCM was originally developed to interface with Xilinx 7-Series devices, but has since been extended to work with earlier Virtex devices as well as compatibility with UltraScale, and UltraScale+ FPGAs. Compatibility with both the space-grade Virtex 5QV FPGA and the Zynq Ultra-Scale+ MPSoC have been of special interest to both government and industry due to their high reliability and potential for use in aerospace applications.
Although this thesis focused completely on interaction with Xilinx FPGAs, there is considerable interest in extending JCM support to Altera\(^1\) devices as well. Although the interface for accessing the CRAM of Intel Altera devices is very different from Xilinx, the layered architecture of the JCM Software Library makes it so that supporting Intel Altera devices will only require modifications to the two top-most layers. Intel Altera FPGAs are less frequently used in aerospace applications; however, they have great potential to be used more widely, and JTAG’s universality makes it a good interface for cross-vendor compatibility.

Another focus of future work will be adding functionality to the JCM to support the Xilinx SelectMAP interface (See Appendix B). This interface is not as universal as JTAG and is specific to only Xilinx devices; however, it is a high-bandwidth parallel interface that can stream as many as 32 bits of data at once. For applications that support SelectMAP, its use would see a large performance benefit over JTAG.

Fault injection has been a primary use of the JCM from the beginning, but has always been limited to the injection of a single DUT. Future work with fault injection will include the ability to use a single JCM to run fault injection campaigns on an array of target JTAG chains. This is important since systematic fault injection campaigns can take long periods of time and the ability to parallelize this process would speed up the gathering of data significantly.

Another focus of future work is extending support of the JCM to non-FPGA devices. Although the JCM already does support some processor debugging functionality, the focus has been on interaction with soft processors implemented in FPGA fabric. Other potential targets include PowerPC and ARM-based hard processors.

The JCM’s success is due in part to the fact that FPGAs are playing an increasingly important role in aerospace applications requiring high performance, flexibility, and reliability. Tools are needed that aid in the development and testing of mechanisms that increase FPGA reliability. Such tools need to be fast and programmable, making the JCM an important contribution to this field of research and development.

\(^1\) Altera was purchased by Intel in 2015 and is also known as the Intel Programmable Solutions Group (PSG).
Bibliography


Appendix A

Using JTAG with FPGAs

Of the many possible applications of JTAG, this work focuses on interaction with Xilinx FPGAs as an example of when high speed and programmability are important. This appendix will provide the background necessary to understand why high speed JTAG is valuable by describing in detail the Xilinx Configuration Registers and how they are accessed. The first section will briefly summarize the main configuration registers, while the second section will discuss the mechanism used by JTAG to access these registers. This appendix only scratches the surface of all that goes into real Xilinx CRAM interaction, but is enough to understand the mechanisms behind the abstractions that the JCM software library provides.

A.1 Xilinx Configuration Registers

Xilinx FPGAs have a set of configuration registers that serve as the external interface to all interaction with the FPGA configuration memory. This set includes the status register, the configuration options registers, the command register, the registers used to read and write to the configuration memory, and many others. A full list of Xilinx 7-Series configuration registers is shown in Figure A.1. This section will describe the subset of registers that are relevant to the scope of this thesis. Further information on these and other registers can be found in [1].

A.1.1 Simple Configuration Registers

Most configuration registers contain single 32-bit values that maintain state information, carry out commands, or control options. Such registers include the Status register, the Frame Address register (FAR), and the ID Code register. The two Xilinx configuration memory access registers (FDRI and FDRO) are exceptions to the rule in that they are variable-length portals to the configuration memory. Those registers that are relevant to JCM applications are the focus of this and the following sections.

The first important register to JCM applications is the Status register. This read-only register contains valuable information about the status of the configuration memory. Among other things it indicates whether the FPGA has been properly programmed, whether initialization has completed, and whether a configuration upset caused by radiation or fault injection has occurred. This register is also frequently read to determine whether a configuration memory transaction was completed successfully.

Another important register is the Frame Address Register (FAR). The FAR contains an address specifying a particular portion of the configuration memory. While a given address
Figure A.1: Xilinx 7-Series Configuration Registers [1]

is held in the FAR, the corresponding memory segment is the only one that can be read or written. This register will be discussed in greater detail in the following subsection.

The next important register is the ID code register. It holds the device ID code which is a 32-bit number that encodes the device manufacturer ID, the family, and the version. Reading this ID code is useful for determining whether communication with the FPGA is working properly.
Other important registers include the Command register, the Control registers, and the Configuration Options registers. These perform important functions such as determining what CRAM data is visible, controlling the internal memory scan (RBCRC), and preparing the device for memory access.

A.1.2 Configuration Memory Access

Access to the FPGA configuration memory is performed by writing to the Frame Data Input register (FDRI) or reading from the Frame Data Output register (FDRO). The Frame Address Register (FAR) that was introduced in the previous subsection specifies where in memory the data is written to or read from. These registers allow users to access any part of the configuration memory. This capability is essential to many applications that seek to monitor, debug, scrub, or modify the configuration memory of FPGAs.

The configuration memory is subdivided into units called frames, with each frame consisting of a few thousand bits of memory. It is the smallest unit of memory that can be read or written. Each frame has a unique address that indicates its physical location as well as its configuration type. This is the address that is contained in the FAR when reading or writing. Table A.1 shows how the frame address of 7-Series FPGAs is used to indicate type and location.

<table>
<thead>
<tr>
<th>Address Type</th>
<th>Bit Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Type</td>
<td>[25:23]</td>
<td>Valid block types are CLB, I/O, CLK (000), block RAM content (001), and CFG_CLB (010). A normal bitstream does not include type 011.</td>
</tr>
<tr>
<td>Top/Bottom Bit</td>
<td>22</td>
<td>Select between top-half rows (0) and bottom-half rows (1).</td>
</tr>
<tr>
<td>Row Address</td>
<td>[21:17]</td>
<td>Selects the current row. The row addresses increment from center to top and then reset and increment from center to bottom.</td>
</tr>
<tr>
<td>Column Address</td>
<td>[16:7]</td>
<td>Selects a major column, such as a column of CLBs. Column addresses start at 0 on the left and increase to the right.</td>
</tr>
<tr>
<td>Minor Address</td>
<td>[6:0]</td>
<td>Selects a frame within a major column.</td>
</tr>
</tbody>
</table>

As discussed in the previous section, the FAR contains the frame address of the currently selected memory frame. Any data written to the FDRI will be written to the specified frame, and data read out of the FDRO also comes from this frame. By default, as soon as a full frame of data is read or written to one of the frame data registers, the address in the FAR auto-increments to the next valid frame address. This allows large amounts of contiguous memory to be read or written by simply writing the starting frame address to
the FAR once, and then performing the large read or write transaction without needing to update the FAR. This auto-incrementation also allows for the creation of a list of all valid frame addresses by reading the FAR each time after a frame of data is read.

A.2 Accessing the Xilinx Configuration Registers

This section will describe in detail the procedure used to access the Xilinx configuration registers. An understanding of this procedure will highlight the importance of a software library that successfully abstracts away such complexity for the user. Much of the content presented in this section as well as further information can be found in [1].

The basic unit of communication with Xilinx configuration registers is a configuration packet. These packets are delivered via JTAG or some other configuration interface to an internal controller that parses the packets and carries out the packet’s indicated operation. Each packet consists of a header section and a data section.

Table A.2: Type 1 Configuration Packet Format [1]

<table>
<thead>
<tr>
<th>Header Type</th>
<th>Opcode</th>
<th>Register Address</th>
<th>Reserved</th>
<th>Word Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:29]</td>
<td>[28:27]</td>
<td>[26:13]</td>
<td>[12:11]</td>
<td>[10:0]</td>
</tr>
<tr>
<td>001</td>
<td>xx</td>
<td>RRRRRRRRRxxxxx</td>
<td>RR</td>
<td>xxxxxxxxxx</td>
</tr>
</tbody>
</table>

The header, shown in Figure A.2, is a 32-bit word that gives meaning to the data that follows. The 32 header bits are divided into sections that each communicate different information. One section specifies which configuration register should be read or overwritten while another indicates how many words of data the packet contains. Since the “Word Count” section of the typical packet header only contains 11 bits, only 2047 words can be sent in a single packet. In cases where this number is insufficient, there is a second type of packet header, shown in Figure A.3 that has 27 bits to specify the number of words to be sent. This second type of packet must be preceded by the first type to specify the source or destination of the data.

Table A.3: Type 2 Configuration Packet Format [1]

<table>
<thead>
<tr>
<th>Header Type</th>
<th>Opcode</th>
<th>Word Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:29]</td>
<td>[28:27]</td>
<td>[26:0]</td>
</tr>
<tr>
<td>010</td>
<td>RR</td>
<td>xxxxxxxxxxxxxxxxxxxxxxxxx</td>
</tr>
</tbody>
</table>
The data section contains the data that will be written to the specified configuration register or to memory. This data can range in size from a single 32-bit word to as many as 134,217,728 words. For example, at the beginning of a typical configuration bitstream, there are several 1-word configuration packets followed by a single packet containing the entire FPGA configuration memory in its data section.

These configuration packets along with certain special words are the substance of all communication with the Xilinx configuration registers. Often, multiple configuration packets must be sent together to successfully complete a more complex operation such as reading or writing a frame of configuration data. One or more configuration packets sent together will hereafter be referred to as a command sequence.

Each command sequence contains configuration packets as well as special words that are used for synchronization and for meeting certain timing constraints specific to the configuration interface. All command sequences begin with two special words called the “dummy word” and the “sync word”. These special words announce the beginning of a command sequence to the target device. A third special word called a “no operation” or “noop” is also frequently used throughout a command sequence to allow previous configuration packets to take effect before the next one is read in. These NOOPs are used less frequently with the JTAG interface since it is by nature much slower than other parallel high-bandwidth interfaces and thus risks violating fewer timing constraints.

Table A.4: A Command Sequence to Write to Configuration Memory

<table>
<thead>
<tr>
<th>Index</th>
<th>Command Word</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xFFFFFFFF</td>
<td>Dummy Word</td>
</tr>
<tr>
<td>1</td>
<td>0xAA995566</td>
<td>Sync Word</td>
</tr>
<tr>
<td>2</td>
<td>0x20000000</td>
<td>No Operation (NOOP)</td>
</tr>
<tr>
<td>3</td>
<td>0x30008001</td>
<td>Packet Type 1: Write to the Command Register</td>
</tr>
<tr>
<td>4</td>
<td>0x00000007</td>
<td>Command: Reset the CRC</td>
</tr>
<tr>
<td>5</td>
<td>0x30018001</td>
<td>Packet Type 1: Write to the ID Code Register</td>
</tr>
<tr>
<td>6</td>
<td>0x0484A093</td>
<td>Value: Device ID Code</td>
</tr>
<tr>
<td>7</td>
<td>0x30008001</td>
<td>Packet Type 1: Write to the Command Register</td>
</tr>
<tr>
<td>8</td>
<td>0x00000001</td>
<td>Command: Write to the configuration memory</td>
</tr>
<tr>
<td>9</td>
<td>0x20000000</td>
<td>No Operation (NOOP)</td>
</tr>
<tr>
<td>10</td>
<td>0x30002001</td>
<td>Packet Type 1: Write to the Frame Address Register (FAR)</td>
</tr>
<tr>
<td>11</td>
<td>0x00000005</td>
<td>Beginning Frame Address: 5</td>
</tr>
<tr>
<td>12</td>
<td>0x20000000</td>
<td>No Operation</td>
</tr>
<tr>
<td>13</td>
<td>0x30004000</td>
<td>Packet Type 1: Write to the Frame Data Register In (FDRI)</td>
</tr>
<tr>
<td>14</td>
<td>0x500013BA</td>
<td>Packet Type 2: Write 5050 words of data (13BA in hex)</td>
</tr>
<tr>
<td>15-5064</td>
<td>(Data)</td>
<td>Value: The data to be written to frames 5 through 54</td>
</tr>
</tbody>
</table>
Figure A.4 shows a command sequence that writes 50 frames of data to the configuration memory. Like all command sequences, it starts with a dummy word, a sync word, and a noop. It then performs a necessary sequence of writes to the command register, the ID code register, the frame address register, and finally the frame data register. All packets in this command sequence are type 1 except for the last packet, which contains the large array of configuration memory data to be written. This final packet contains both a type 1 and type 2 header so that it can write more than the 2047 word limit of the type 1 packet.

Once a command sequence is assembled, JTAG is used to send it to the configuration register controller. Each command sequence is streamed into a special Xilinx JTAG-accessible register called CFG_IN. This is done by putting the CFG_IN register address into the JTAG Instruction Register (IR), and then shifting the command sequence into the JTAG Data Register (DR). Then, if the command sequence was a read request, the requested data can be read out of another special JTAG-accessible register called CFG_OUT.

### A.2.1 Accessing Other JTAG-Accessible FPGA Registers

Besides providing access to the FPGA configuration memory with its CFG_IN and CFG_OUT registers, JTAG can also expose or modify internal logic signals through its four USER registers. The designer simply attaches internal signals to one of the four available BSCAN primitives, and then writes or reads out the attached signal values using the corresponding USER register through JTAG. This feature can be useful for FPGA design monitoring and debugging.

Another useful JTAG register is the XADC_DRP, which is the portal to communicate with the Xilinx Analog to Digital Converter. This register allows the user to read the on-chip temperature and voltages.
Appendix B

FPGA Configuration Interfaces

Besides JTAG, there exist several other Xilinx FPGA configuration interfaces. These include the Internal Configuration Access Port (ICAP), the Processor Configuration Access Port (PCAP), and SelectMAP. Each of these interfaces has its pros and cons as well as a set of applications for which it is most effective. This thesis will endeavor to show that with the proper infrastructure to mitigate its drawbacks, JTAG is an excellent interface for a wide variety of FPGA applications.

B.1 ICAP

The Internal Configuration Access Port (ICAP) is a hardware IP core which allows a soft processor or hardware state machine to interact with the configuration registers of the FPGA upon which it is implemented. To use it, the ICAP primitive must be instantiated in the hardware design (see Figure B.1). The ICAP cannot perform the initial FPGA configuration, but once the FPGA has been successfully programmed, the ICAP allows a user design to either partially or fully reconfigure the FPGA [21] [33].

![Figure B.1: ICAPE2 Configuration Interface [3]](image)

One of the main advantages of the ICAP as a configuration interface is its speed, reaching up to 100 MHz for some devices [34]. Since it is a parallel interface with 8, 16, or 32 data bits, the ICAP can achieve configuration speeds of 3.2 Gb/s. One disadvantage for reliability applications is that since the ICAP is an FPGA primitive, it is just as sensitive to
radiation upsets as the FPGA design that it is designed to protect. When used for scrubbing in a radiation test, it frequently fails due to radiation upsets.

Several academic tools using the ICAP have been developed. The tool described in [33] consists of a Virtex 4 FPGA with a PicoBlaze soft processor design that uses the ICAP to perform scrubbing on itself. It uses the built-in Error Correction Code (ECC) logic to detect and correct single-bit errors. It solves the problem of the ICAP itself being sensitive by triplicating the ICAP circuit and using block memory scrubbing.

Another tool, described in [35], also uses the ICAP in a Virtex-5 FPGA to self scrub its CRAM. It goes beyond single-error correction, double-error detection (SECDED) capability by using a new technique called Frame-Level Redundancy (FLR) Scrubbing. This is done by performing course-grained triple-modular redundancy (TMR) on the design so that three copies of each frame exist, and then using a voter circuit to perform bit-level comparison. The ICAP is used to correct the errors found in any of the three frames. This technique works as long as the ICAP itself does not fail and the same bit in multiple identical memory frames is not upset.

B.2 PCAP

The Processor Configuration Access Port (PCAP) is a processor peripheral found only on the Zynq-7000 and the Zynq UltraScale+ MPSoC parts. It provides configuration access to the on-chip ARM processing system. This allows software running on the processor rather than the FPGA design itself to handle the interaction with the Xilinx configuration registers. Unlike the ICAP, the PCAP can perform the initial FPGA configuration itself, before going on to perform partial or full reconfiguration [21].

The PCAP is also known for its high speed. It also can operate at speeds of 100 MHz, which translates to a 3.2 Gb/s configuration speed with its 32-bit data bus [36]. Unfortunately, like the ICAP, it also suffers from the same reliability issues as the FPGA design that it may be helping to protect, since they are both part of the same SoC.

The tool described in [21] uses the PCAP of a Zynq-7000 SoC to maximize the speed and robustness of error detection and correction. It uses the Frame_ECC primitive to monitor the status of the internal Readback CRC scan, which achieves SECDED functionality, and jumps in to scrub whenever a multi-bit upset occurs. Although this tool is extremely fast and highly robust at scrubbing the FPGA, radiation-induced processor failures remain a significant reliability concern.

B.3 SelectMAP

SelectMAP is a Xilinx-specific external configuration interface. As an external interface, SelectMAP does not utilize any internal circuitry, but rather the configuration registers are accessed directly through I/O pins of the FPGA. This allows an external controller to access the CRAM of the FPGA. SelectMAP is also able to perform the initial configuration as well as perform partial or full reconfiguration after the FPGA has been programmed.

Like the ICAP and PCAP, SelectMAP is also a parallel high-bandwidth interface. It reaches operating speeds of 100 MHz and can be configured to have an 8, 16, or 32-bit bidirectional data bus [1]. Unlike the two internal configuration interfaces, SelectMAP does not share radiation sensitivity with the FPGA design that it may help to protect; however, the
external controller using SelectMAP may have its own sensitivity issues. Another drawback to SelectMAP is the need for a large number of I/O pins to support the SelectMAP data and control signals [22].

The techniques described in [37] use the SelectMAP interface for high-speed configuration scrubbing. This paper recommends planning for reliability mechanisms in the design phase so that the necessary infrastructure for maximal reliability can be built in from the beginning. It also asserts that for many applications, the SelectMAP interface, when available, is preferable because of its speed and ability to both initially configure and reconfigure an FPGA at runtime.

B.4 JTAG

Unlike its counterparts, JTAG is a light-weight external serial interface. Like SelectMap, it requires external I/O pins to operate, but since it is a serial interface, it only needs 4 of them [15]. Also, since it is a universal standard for testing, it is already built into all Xilinx FPGA boards, making it easy to use. It is also always available to be used for configuration, regardless of what configuration mode the device may be in [1].

The main drawback to JTAG is its slower speed. With a maximum speed of about 66 MHz and only a single data line, JTAG can only configure at up to 66 Mb/s [38]. Furthermore, most JTAG programmers operate at less than half of the maximum speed due to device and cable constraints [39] [40].

Despite being slower than other interfaces, JTAG’s universality, low overhead, and small radiation-sensitive cross section make it the interface of choice for a variety of common configuration tools from both industry and academia [14]. Both Xilinx and Digilent provide USB-to-JTAG translation modules as well as software APIs for performing custom JTAG transactions on Xilinx FPGAs. Although these are sufficient for basic applications, they are limited in speed to between 10-30 MHz and require in-depth understanding of JTAG and the Xilinx configuration interface for more advanced applications. Among the various recent academic configuration tools that use JTAG is the one described in [13]. It is a software based tool that provides an easy interface for using the low-level Xilinx JTAG API for fault injection on a Virtex 5 FPGA. Another, described in [14], is an embedded processor running software with an FPGA based JTAG controller for scrubbing Virtex 4 FPGAs. It achieves a maximum JTAG speed of 25 MHz.

One goal of this thesis is to explain the mechanisms that enable the JCM to maximize the speed of JTAG configuration. It also seeks to highlight the benefits of JTAG for certain applications that make it the interface of choice despite its speed limitations.
Appendix C

JCM Software Application Programming Interface (API)

This appendix contains the application programming interface (API) for the JCM software library. This software library was designed with custom development in mind. Although this thesis has focused on several specific applications of the JCM that are already supported by the JCM Software Library, this application programming interface allows for adaptation of the JCM for other purposes.

This appendix consists of the auto-generated web documentation for the XilinxTopLibrary class. This is the primary top-level API for the JCM; however, for development using the lower levels of the software architecture, this documentation is insufficient. More detailed documentation exists, but is beyond the scope of this thesis.

C.1 Xilinx Top Library API Methods
# include `<XilinxTopLibrary.h>`

## Public Member Functions

- **XilinxTopLibrary** (string jcmConfigFile="", string chainConfigFile="")
- **XilinxTopLibrary** (HardwareConfig *hwConfig, ChainConfig *chainConfig)
- void **init** (HardwareConfig *hwConfig, ChainConfig *chainConfig)
- void **printHardwareVersion** ()
- u32 * **readFullDevice** (std::string fileName, bool readBram)
- void **setActiveDeviceIndex** (unsigned int index)
- u32 * **initializeScrubber** (bool performInitReadback, bool readBram, bool disableInternalScan, bool restoreOldSession, string goldenFile)
- int **calibrateDevice** (bool fullSweep, int targetSpeed, string chainConfigFile, bool updateConfigFile)
- void **updateConfigFile** (string fileName, int numExtraShifts)
- int **compareFrameData** (int numFrames, int numWordsPerFrame, u32 *testData, u32 *goldenData, bool scrubErrors, bool verbose, u32 startingFrad)
- u32 **readFar** ()
- u32 **readIdCode** ()
- u32 **readCtrl0** ()
- u32 **readCrc** ()
- u32 **readCrcHw** ()
- u32 **readCrcSw** ()
- u32 **readCrcLive** ()
- u32 **readStatus** ()
- u32 **readCor1** ()
- u32 **readCmd** ()
- float **readXadcCurTemp** ()
- float **readXadcVccInt** ()
- float **readXadcVccAux** ()
- float **readXadcTemp** (u32 readCommand)
- float **readXadcVoltage** (u32 readCommand)
- u32 * **readFrames** (u32 beginFrameAddress, int numFrames)
- u32 * **readBscan** (int bscanNumber, int bscanNumBits)
- void **writeBscan** (int bscanNumber, int numWordsToWrite, u32 *regValue)
- void **writeFar** (u32 regValue)
- void **writeCor1** (u32 regValue)
void writeCrcSw (u32 regValue)
void clearGlutMaskBit ()
void setGlutMaskBit ()
void issueProg (u32 WBStarAddr)
void issueCapture ()
void clearCmd ()
void fullDeviceConfigure ()
bool checkValidFrameAddress (u32 address)
int getFrameAddressIndex (u32 address, u32 *addressArray, int arraySize)
bool injectRandomFault (int faultInjectionSize, bool verifyInjection, bool repairFault, bool verbose)
void injectMultiFrameFault (u32 frad, u32 command)
bool injectFault (u32 frameAddress, int wordNum, int bitNum, int numBits, bool printData, bool verifyInjection)
void blindScrub (bool performInitReadback)
void readbackScrub (bool performInitReadback, bool readBram, bool fixErrors, bool saveToFile, int milliSecDelay, bool printErrors, bool overwriteGolden, bool injectFaults, string maskFile)
void frameBasedScrubbing (bool performInitReadback, bool readBram, bool fixErrors, bool verbose)
int detectStuckBits (bool performInitReadback, std::string outputFileName)
void hybridScrub (bool performInitReadback, bool fullReadbackOnCrcError, int faultInjectionSize, bool reconfigure, string goldenFile)
void simpleHybridScrub (bool performInitReadback, bool correctionChecking, bool fullReadbackOnCrcError, int faultInjectionSize)
int getWordsPerFrame ()
int getNumLogicFrames ()
int getNumBramFrames ()
int getNumLogicAndBramFrames ()
ChainConfig * getChainConfig ()
int getNumBitFileWords ()
u32 * getFrameAddressArray ()
void setFrameAddressArray (u32 *fradArray)
FradStructure * getFradStructure ()
void setFradStructure (FradStructure *fradStructure)
XilinxVendor * getVendor ()
XilinxProcessor * getXilinxProcessor ()
HardwareConfig * getHardwareConfig ()
XilinxFaultInjection * getFaultInjector ()
XilinxReadback * getReadback ()
Constructor & Destructor Documentation

XilinxTopLibrary::XilinxTopLibrary ( string jcmConfigFile = "",  
                                  string chainConfigFile = ""  
                                  )

This constructor takes a jcmConfig file name and a chain configuration file name. It creates a **HardwareConfig** object and a **ChainConfig** object using the config files and then calls the other constructor

**Parameters**

- **jcmConfigFile**  The name of the jcmConfigFile. If an empty string is passed ("") then it looks for the file in the default location ("/root/jcm/configFiles/jcm_config.txt")
- **chainConfigFile**  The name of the chain config file. If an empty string is passed ("") then it uses the file name provided in the jcmConfig.txt file.

**Returns**

A pointer to the **XilinxTopLibrary** object

XilinxTopLibrary.cpp

This class provides a high level library for interaction with the Xilinx configuration registers and memory

**Author**

Ammon Gruwell

XilinxTopLibrary::XilinxTopLibrary ( HardwareConfig * hwConfig,  
                                    ChainConfig * chainConfig  
                                    )

This constructor takes a **HardwareConfig** object (specifies HW type and base address) and a **ChainConfig** object (contains JTAG chain info)

**Parameters**

- **hwConfig**  The **HardwareConfig** object that specifies HW type and base address
- **chainConfig**  The **ChainConfig** object that contains JTAG chain information

**Returns**

A pointer to the **XilinxTopLibrary** object
void XilinxTopLibrary::blindScrub ( bool performInitReadback )

This function performs a Blind scrub on the configuration memory. It continuously writes the golden data to the device without checking for errors in the config memory.

**Parameters**

performInitReadback This specifies if an initial readback of the whole device is desired

int XilinxTopLibrary::calibrateDevice ( bool fullSweep,
                                      int targetSpeed,
                                      string chainConfigFile,
                                      bool updateConfigFile )

Calibrates the JTAG hardware state machine to work with the current cable length and target device type.

**Parameters**

fullSweep If TRUE, test the full range of clock speeds to find the various speed boundaries

targetSpeed If non-zero, indicates the specified clock speed if possible.

chainConfigFile The name of the chain configuration file, if an empty string ("") is provided then the config file is not updated

updateConfigFile Specify if the chain configuration file should be updated. This is always false when using the python interface, but true otherwise

**Returns**

The number of extra shifts needed if calibration was successful, -1 otherwise

bool XilinxTopLibrary::checkValidFrameAddress ( u32 address )

This function checks if the provided address is a valid address of the active device

**Returns**

true if the address is valid

void XilinxTopLibrary::clearCmd ( )

This function clears the command register
void XilinxTopLibrary::clearGlutMaskBit()

This function clears the GLUT Mask Bit in the CTRL0 register to prevent reading dynamic bits back from the configuration memory.

int XilinxTopLibrary::compareFrameData(int numFrames, int numWordsPerFrame, u32 * testData, u32 * goldenData, bool scrubErrors, bool verbose, u32 startingFrad)

Compares two arrays of configuration memory data and scrubs the errors if desired. The number of differences is returned.

Parameters
- **numFrames** - The number of frames to be compared
- **numWordsPerFrame** - The number of words in a single config frame.
- **testData** - The data to be compared with the golden data
- **goldenData** - The array of golden data
- **scrubErrors** - Specifies if the detected errors are fixed
- **verbose** - Specifies if error info should be printed
- **startingFrad** - The starting frame address of the data

int XilinxTopLibrary::detectStuckBits(bool performInitReadback, std::string outputFileName)

This function performs a word by word check for stuck bits. The output will also include masked bits, and therefore will only be useful when compared to another device that is known to contain no stuck bits.

Parameters
- **performInitReadback** - Specifies if an initial readback of the whole device is desired.
- **outputFileName** - The name of the file where results are stored.
void XilinxTopLibrary::frameBasedScrubbing ( bool performInitReadback,  
        bool readBram,  
        bool fixErrors,  
        bool verbose  
    )

This function performs a frame by frame Readback scrub on the config memory. It continuously reads the configuration memory and checks it against the golden data to find errors. When an error is detected the frame is overwritten with the corresponding golden data frame.

Parameters

performInitReadback Specifies if an initial readback of the whole device is desired.
readBram If TRUE, reads all BRAM frames in addition to LOGIC frames.
fixErrors If TRUE, scrubs by writing the entire device's configuration memory (excluding BRAM frames).
verbose If TRUE, print location of errors found.

void XilinxTopLibrary::fullDeviceConfigure ( )

This function configures the device with the bit file read in from the Configuration file.

ChainConfig * XilinxTopLibrary::getChainConfig ( )

CppUtils * XilinxTopLibrary::getCppUtils ( )

XilinxFaultInjection * XilinxTopLibrary::getFaultInjector ( )

FradStructure * XilinxTopLibrary::getFradStructure ( )

u32 * XilinxTopLibrary::getFrameAddressArray ( )

```cpp
int XilinxTopLibrary::getFrameAddressIndex ( u32 address,
                                           u32 * addressArray,
                                           int arraySize
)
```

This function returns the index of the frame address in the array of frame addresses provided.

**Parameters**
- `address` the frame address to translate to an index
- `addressArray` the array of frame addresses
- `arraySize` the size in words of the address array

**Returns**
- the index of the frame address in the array if it is present, otherwise returns -1

```cpp
HardwareConfig * XilinxTopLibrary::getHardwareConfig ( )
```

```cpp
int XilinxTopLibrary::getNumBitFileWords ( )
```

```cpp
int XilinxTopLibrary::getNumBramFrames ( )
```

```cpp
int XilinxTopLibrary::getNumLogicAndBramFrames ( )
```

```cpp
int XilinxTopLibrary::getNumLogicFrames ( )
```

```cpp
XilinxReadback * XilinxTopLibrary::getReadback ( )
```

```cpp
XilinxVendor * XilinxTopLibrary::getVendor ( )
```

```cpp
int XilinxTopLibrary::getWordsPerFrame ( )
```

```cpp
XilinxProcessor * XilinxTopLibrary::getXilinxProcessor ( )
```

```cpp
XilinxUtils * XilinxTopLibrary::getXilinxUtils ( )
```
This function performs a hybrid scrub on the configuration memory. It continuously reads the output of the FRAME_ECC module through a bscan register, and when an error is detected the memory is scrubbed. Requires jtag_scrubber_hw.vhd in your design for use

**Parameters**

- **performInitReadback**: Specifies if initial readback of the whole device is desired.
- **fullReadbackOnCrcError**: If TRUE, performs a full readback of configuration frames on CRC errors and saves the data to a file.
- **faultInjectionSize**: The number of pseudo-random faults to inject in each cycle.
- **reconfigure**: If true, check for failure (LEON3 TMR tests) and reconfigure if error encountered.
- **maskFile**: File name of mask file containing bits to ignore in error checking.
u32 * XilinxTopLibrary::initializeScruber ( bool performInitReadback,
    bool readBram,
    bool disableInternalScan,
    bool restoreOldSession,
    string goldenFile
 )

Initializes the scrubber by clearing GLUT bit, then performing actions specified by the parameters below.

Parameters
- **performInitReadback** If TRUE, perform an initial readback of the entire device and save it to file.
- **readBram** If TRUE, include BRAM frames in the initial readback of the device.
- **disableInternalScan** If TRUE, disable the internal EEC calculation and fault correction of the device.
- **restoreOldSession** Specifies if previous readback session should be restored (generally

Returns
Return a pointer to the device's data.

bool XilinxTopLibrary::injectFault ( u32 frameAddress,
    int wordNum,
    int bitNum,
    int numBits,
    bool printData,
    bool verifyInjection
 )

This injects a fault into the desired configuration frame

Parameters
- **frameAddress** The frame which will receive the fault injection
- **wordNum** The word to inject a fault
- **bitNum** The bit to inject a fault
- **numBits** The number of consecutive bits to upset
- **verifyInjection** When true, this function will also check if the injection succeeded

Returns
TRUE if fault injection was successful or FALSE if it failed.
void XilinxTopLibrary::injectMultiFrameFault ( u32 frad,
            u32 command
        )

Injests up to 6 frames with faults according to which bits are set in the command word. See the #define values for which bits to set in order to inject faults of different types.

**Parameters**

- `frad` Address to inject first fault. Ensure that there are enough logic frames after this address to inject the remaining faults.
- `command` Command word that specifies what types of faults to inject. See #define values for information on bits to set.

bool XilinxTopLibrary::injectRandomFault ( int faultInjectionSize,
                                          bool verifyInjection,
                                          bool repairFault,
                                          bool verbose
                                      )

Injests a fault of the specified size in a random frame, word, and bit location.

**Parameters**

- `faultInjectionSize` Number of bits to inject.
- `verifyInjection` Specify whether or not to verify that the injection was successful.

void XilinxTopLibrary::issueCapture ( )

This function issues a GCAPTURE command.

void XilinxTopLibrary::issueProg ( u32 WBStarAddr )

This function issues a PROG command.

void XilinxTopLibrary::printHardwareVersion ( )

This function prints out the hardware version.
void XilinxTopLibrary::readbackScrub ( bool performInitReadback,
    bool readBram,
    bool fixErrors,
    bool saveToFile,
    int milliSecDelay,
    bool printErrors,
    bool overWriteGolden,
    bool injectFaults,
    string maskFile )

This function performs a Readback scrub on the configuration memory. It continuously reads the configuration memory and checks it against the golden data to find errors. When an error is detected the frame is overwritten with the corresponding golden data frame.

Parameters

performInitReadback Specifies if an initial readback of the whole device is desired.
readBram If TRUE, reads all BRAM frames in addition to LOGIC frames.
fixErrors If TRUE, scrubs by writing the entire device's configuration memory (excluding BRAM frames).
saveToFile If TRUE, saves the readback data to file.
milliSecDelay If nonzero, usleep for (milliSecDelay*1000)
printErrors If TRUE, print location of errors found.
overWriteGolden If TRUE, errors will be saved to the golden file
injectFaults If TRUE, inject faults at pseudo-random locations
continuous If TRUE, readback will occur in an infinite loop until stopped by the user, if FALSE, the readback loop depends upon an external program (GUI) to restart the loop
isFirstIteration Specifies whether this is the first time an external program has called the function (only applicable when "continuous" is false)
```
u32 * XilinxTopLibrary::readBscan ( int bscanNumber, 
    int bscanNumBits 
) 

This function reads the current value of the BSCAN register

Parameters
    bscanNumber  The index of the desired BSCAN register
    bscanNumBits The size in bits of the BSCAN register

Returns
    The current value contained in the BSCAN register

u32 XilinxTopLibrary::readCmd ( )

u32 XilinxTopLibrary::readCor1 ( )

This function reads the contents of the Xilinx COR1 Register which contains information such as whether the internal scan is on or whether ECC calculation is on

Returns
    The current bits of the Status Register

u32 XilinxTopLibrary::readCrc ( )

This function reads the contents of the Xilinx CRC register which holds the current CRC calculation as it is calculating it. This register should match the CRCHW when it reaches the end of the device memory or else a CRC Error is reported

Returns
    The current value in the CRC register

u32 XilinxTopLibrary::readCrcHw ( )

This function reads the contents of the Xilinx CRC Hardware Register which is used by the internal scrubber to save the value of the CRC Register after each pass through the device

Returns
    The current value in the CRC HW register
```
### u32 XilinxTopLibrary::readCrcLive()

This function reads the contents of the Xilinx CRC Live Register which contains the current CRC calculation when the CRC Software is enabled. This and the CRC Software register are compared to detect a CRC Error

**Returns**
- The current value in the CRC Live register

### u32 XilinxTopLibrary::readCrcSw()

This function reads the contents of the Xilinx CRC Software Register which can be enabled for use instead of the CRC Hardware. Unlike the CRC Hardware it is accessible by the user. The user can save the value in the CRC Live to this register after a full pass through of the device

**Returns**
- The current value in the CRC Software register

### u32 XilinxTopLibrary::readCtrl0()

This function reads the contents of the Xilinx CTRL0 Register which contains various bits such as the GLUT mask bit

**Returns**
- The current CTRL0 register contents

### u32 XilinxTopLibrary::readFar()

This function reads the contents of the Xilinx Frame Address Register

**Returns**
- The current Frame Address in the register
**u32** XilinxTopLibrary::readFrames ( **u32** beginFrameAddress, **int** numFrames )

This function reads the desired number of frames from the configuration memory

**Parameters**

- **beginFrameAddress** The frame address at which to start reading
- **numFrames** The number of frames to read

**Returns**

The current internal VCC value

---

**u32** XilinxTopLibrary::readFullDevice ( **std::string** fileName, **bool** readBram )

This function reads the entire device memory and writes it into a file

**Parameters**

- **fileName** The name of the file to save the config memory
- **readBram** Specifies if the bram memory should be read as well

**Returns**

A pointer to the array of device data

---

**u32** XilinxTopLibrary::readIdCode ( )

This function reads the contents of the Xilinx ID Code Register

**Returns**

The device ID code

---

**u32** XilinxTopLibrary::readStatus ( )

This function reads the contents of the Xilinx Status Register which contains among other things the DONE pin and the CRC Error bits

**Returns**

The current bits of the Status Register
float XilinxTopLibrary::readXadcCurTemp()

This function reads the current temperature through the Xilinx XADC.

Returns
The current die temperature

float XilinxTopLibrary::readXadcTemp(u32 readCommand)

This function reads the specified temperature register through the Xilinx XADC.

Returns
The specified temperature reading

float XilinxTopLibrary::readXadcVccAux()

This function reads the auxiliary voltage through the Xilinx XADC.

Returns
The current auxiliary voltage

float XilinxTopLibrary::readXadcVccInt()

This function reads the internal voltage through the Xilinx XADC.

Returns
The current internal voltage

float XilinxTopLibrary::readXadcVoltage(u32 readCommand)

This function reads the specified voltage level register through Xilinx XADC.

Returns
The specified voltage reading
void XilinxTopLibrary::setActiveDeviceIndex ( unsigned int index )

This function allows the user to change the active device on the JTAG chain dynamically at runtime

**Parameters**

index Index of device to be used

void XilinxTopLibrary::setFradStructure ( FradStructure * fradStructure )

void XilinxTopLibrary::setFrameAddressArray ( u32 * fradArray )

void XilinxTopLibrary::setGlutMaskBit ( )

This function sets the GLUT Mask Bit in the CTRL0 register to unmask dynamic bits in the configuration memory

void XilinxTopLibrary::simpleHybridScrub ( bool performInitReadback,
                                       bool correctionChecking,
                                       bool fullReadbackOnCrcError,
                                       int faultInjectionSize )

This function is a very simple hybrid scrubbing algorithm that is meant to explain the basics of hybrid scrubbing methodology. It has flaws that are addressed in the full-fledged algorithm used in the hybridScrub function. Requires jtag_scrubber_hw_simple.vhd in your design for use

**Parameters**

performInitReadback Specifies if initial readback of the whole device is desired.

correctionChecking If TRUE, checks to see if internal scrubber corrected the error.

fullReadbackOnCrcError If TRUE, performs a full readback of configuration frames on CRC errors and saves the data to a file.

faultInjectionSize The number of pseudo-random faults to inject in each cycle.
void XilinxTopLibrary::updateConfigFile ( string fileName,
    int numExtraShifts
 )

Updates the specified chain configuration file with the given number of extra shifts.

Parameters
fileName The name of the chain configuration file
numExtraShifts The number of extra shifts needed

void XilinxTopLibrary::writeBscan ( int bscanNumber,
    int numWordsToWrite,
    u32 * regValue
 )

This function writes to a bscan register

Parameters
bscanNumber The index of the desired BSCAN register
numWordsToWrite The number of words to write to the register
regValue The words to be written

void XilinxTopLibrary::writeCor1 ( u32 regValue )

This function writes the specified value into the Xilinx COR1 register

Parameters
regValue The value to be written into the register

void XilinxTopLibrary::writeCrcSw ( u32 regValue )

This function writes the specified value into the Xilinx CRC_SW register

Parameters
regValue The value to be written into the register
void XilinxTopLibrary::writeFar ( u32 regValue )

This function writes the specified value into the Xilinx Frame Address Register

Parameters

regValue The Frame Address to be written into the register