

EMULATED FAULT INJECTION FOR BUILT-IN SELF-TEST  
OF FIELD PROGRAMMABLE GATE ARRAYS USING BOUNDARY SCAN

Except where reference is made to the work of others, the work described in this undergraduate thesis is my own or was done in collaboration with my advisor. This thesis does not include proprietary or classified information.

---

Mustafa Ali

Certificate of Approval:

---

Charles E. Stroud  
Professor  
Electrical and Computer Engineering

---

James R. Hansen  
Director  
University Honors College

EMULATED FAULT INJECTION FOR BUILT-IN SELF-TEST  
OF FIELD PROGRAMMABLE GATE ARRAYS USING BOUNDARY SCAN

Mustafa Ali

A Thesis

Submitted to the

Auburn University Honors College

in Partial Fulfillment of the

Requirements for the

University Honors Scholar

Auburn, Alabama  
August 4, 2007

EMULATED FAULT INJECTION FOR BUILT-IN SELF-TEST  
OF FIELD PROGRAMMABLE GATE ARRAYS USING BOUNDARY SCAN

Mustafa Ali

Permission is granted to Auburn University to make copies of this thesis at its discretion, upon the request of individuals or institutions and at their expense. The author reserves all publication rights.

---

Signature of Author

---

Date of Graduation

## VITA

Mustafa Ali, son of Muhammad and Jehanara Ali, was born on July 19, 1985, in Karachi, Pakistan. He obtained an International Baccalaureate diploma from Auburn High School in May 2003. After graduation, Mustafa entered the Bachelor of Electrical Engineering program at Auburn University with a concentration in Computer Engineering and a minor in Business Administration. While pursuing his Bachelors degree, Mustafa worked under the guidance of Dr. Charles E. Stroud as an undergraduate research assistant in the Auburn University Built-In Self-Test (AUBIST) laboratory.

UNDERGRADUATE HONORS THESIS ABSTRACT  
EMULATED FAULT INJECTION FOR BUILT-IN SELF-TEST  
OF FIELD PROGRAMMABLE GATE ARRAYS USING BOUNDARY SCAN

Mustafa Ali

Bachelor of Science, August 4, 2007

82 Typed Pages

Directed by Charles E. Stroud

Increasingly large and complex Field Programmable Gate Arrays (FPGAs) present greater testing challenges and greater costs. Built-In Self-Test (BIST) is an in-system testing procedure for detecting internal faults in an FPGA without area or performance penalty to the system's intended function. The tester downloads and executes BIST configurations on an FPGA, and reads back the results to determine if any faults exist in the FPGA. With large FPGAs, however, it is difficult to experimentally verify if a BIST method accurately detects faults without a controlled method of injecting faults into the FPGA. Moreover, any fault injection method needs to be reversible so as not to render the FPGA unusable.

This thesis discusses a systematic method of reconfiguring FPGAs such that the behavior of the FPGA emulates an actual fault. The proposed fault injection method relies on single-bit partial reconfiguration of an FPGA after each BIST configuration

download so that, during the BIST execution, the FPGA behaves in a manner consistent with a faulty FPGA. The method utilizes the existing Boundary Scan architecture without modifications. Moreover, it does not impose area and design constraints on BIST configurations. Since current FPGAs such as Virtex-4 do not support bit-level partial reconfiguration, the proposed method uses a frame read-modify-write procedure to achieve bit-level partial reconfiguration. The thesis discusses actual experimental results and proposes methods for reducing time penalties in future fault injection methods.

## ACKNOWLEDGEMENTS

I reserve special thanks for Dr. Stroud for his support and mentorship throughout my undergraduate research. His entertaining style in the lab, the classroom, and life is an important reason why I enjoyed working with him from the very beginning. I would also like to thank Dr. Victor Nelson, Dr. Vishwani Agrawal, Dr. A. S. Hodel, and the entire faculty for their years of guidance. In addition, this research endeavor would not have been possible without my colleagues in the Auburn University Built-In Self-Test Lab – Daniel, Bobby, Lee, Chris, Sachin and Sudheer – with their friendship, guidance, and general sanity when FPGAs wouldn't behave.

The opportunities provided by and the support from the University Honors College and Assistant Director Kathie Mattox have been instrumental in my personal and professional development. I also want to thank all my friends, Insomnia, and Skittles for four very exciting years. Most importantly, I would like to express my deepest gratitude to my parents and sister Maryam for years of love and encouragement to pursue my dreams.

So long, and thanks for all the fish!

Style manual or journal used: IEEE (Institute of Electrical and Electronic Engineers) Journal style

Computer software used: Microsoft Office Word 2003

## TABLE OF CONTENTS

LIST OF FIGURES .....	XI
LIST OF TABLES .....	XII
CHAPTER 1 INTRODUCTION .....	1
1.1 Overview of FPGAs.....	2
1.2 Overview of FPGA Testing .....	3
1.3 Verifying BIST Designs using Fault Injection .....	4
CHAPTER 2 BACKGROUND.....	6
2.1 The Xilinx Virtex-4 FPGA Architecture .....	6
2.2 Virtex-4 Configuration Memory Structure .....	10
2.3 Boundary Scan Interface to the Configuration Memory.....	11
2.4 Built-In Self-Test for Virtex-4 FPGA.....	15
2.5 Prior Work in Fault Injection of FPGAs.....	16
2.6 Fault Injection for Verification of Virtex-4 BIST Design .....	18
CHAPTER 3 EMULATION OF FAULT USING PARTIAL RECONFIGURATION .....	20
3.1 Mapping Fault Location to the Configuration Memory Bit.....	21
3.2 Partial Frame Readback .....	22
3.3 Frame Data Modification.....	26
3.4 Partial Frame Reconfiguration.....	27
3.5 Implementation of Fault Emulation in C .....	32
3.6 Experimental Results of Fault Emulation.....	35
CHAPTER 4 INTEGRATION OF FAULT INJECTION INTO THE BIST PROCEDURE.....	38
4.1 Integration Overview .....	38
4.2 Implementation in Virtex-4 BIST GUI.....	40
4.3 Experimental Results .....	45
CHAPTER 5 BOUNDARY SCAN OPERATIONAL TEST .....	46
5.1 Overview of Boundary Scan User-Defined Registers .....	47

5.2	Test Configuration .....	48
5.3	Test Procedure .....	51
5.4	Fault Detection in Boundary Scan USER Registers .....	53
5.5	Fault Detection in the User Access Register.....	57
5.6	Test Results.....	58
CHAPTER 6 SUMMARY AND CONCLUSIONS .....		60
6.1	Summary and Main Contributions.....	60
6.2	Future Research .....	62
BIBLIOGRAPHY .....		65
APPENDICES.....		66
APPENDIX A LIST OF ACRONYMS .....		67
APPENDIX B VIRTEX-4 DEVICE IDS.....		69
APPENDIX C VIRTEX-4 BOUNDARY SCAN INSTRUCTION CODES.....		70

## LIST OF FIGURES

Figure 2.1: FPGA with PLBs, I/O cells, and programmable interconnect .....	7
Figure 2.2: One PLB with four slices [8].....	8
Figure 2.3: 16-state TAP Controller from Virtex-4 configuration guide.....	13
Figure 3.1: Frame RMW performed using Windows-based Application.....	33
Figure 4.1: Software functions to implement BIST with fault injection .....	41
Figure 5.1: USER Register and related test circuitry for Boundary Scan User Module .	49
Figure 5.2: Implementation of a single bit SR(i) inside the USER register .....	51

## LIST OF TABLES

Table 2.1: Boundary Scan Pin Description.....	12
Table 2.2: Boundary Scan operations to access configuration memory interface.....	14
Table 3.1: A frame's address is based on its physical location on the FPGA.....	21
Table 3.2: Boundary Scan signals to load the Boundary Scan CFG_IN instruction.....	23
Table 3.3: Boundary Scan operations for partial reconfiguration.....	28
Table 3.4: Fault Injection Function Source Code.....	34
Table 3.5: 4-Input XOR Truth Table.....	36
Table 4.1: Original BIST process without fault injection.....	39
Table 4.2: Revised BIST process with additional fault injection step.....	39
Table 4.3: BIST process for a large fault list.....	40
Table 4.4: "Auto Fault Inject" function source code.....	42
Table 4.5: "Get Fault from Fault File" function source code.....	43
Table 5.1: Experimental results as expected for LX60, SX35 and FX12.....	59
Table 5.2: Compressed Bitstream Size.....	59
Table 6.1: Number of faults emulated for various BIST approaches.....	62

## CHAPTER 1

### INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are pre-fabricated programmable semiconductor devices whose array of programmable logic and routing can be configured to create complex combinatorial and sequential circuits [1]. FPGAs are becoming increasingly popular in electronic designs due to their re-configurability, and the ease of implementation and verification. Higher-end FPGAs also include dedicated memory, microprocessors, digital signal processors and communication modules to increase their versatility [2].

As FPGAs become larger, faster and more complex, it is increasingly difficult and expensive to exhaustively test all FPGA resources for faults [2]. To address these issues, Built-In Self-Test (BIST) for FPGAs has emerged as a powerful yet cost-effective method for testing an FPGA's internal resources. One indicator of a BIST approach's effectiveness is its fault coverage, which is defined as the ratio of the number of faults detected to the total number of possible faults [3]. Another indicator is the diagnostic resolution of the BIST, that is, how well a BIST can pinpoint the fault based on test results.

With the ever-increasing size of FPGAs, however, there is a need for straightforward experimental verification of a BIST algorithm's performance or fault

coverage. One method of experimentation is the injection of a fault into the FPGA before running a BIST [4]. If the BIST is able to pick up the fault, the test is deemed successful. This thesis describes a systematic method of injecting faults in an FPGA to determine a BIST algorithm's performance or fault coverage.

## **1.1 Overview of FPGAs**

FPGAs are primarily an array of programmable logic blocks (PLBs), a routing matrix and I/O cells [1]. The PLBs typically contain programmable Look Up Tables (LUTs), which store multi-input logic functions, and flip-flops (FFs). The routing matrix, also known as the programmable interconnect network, consists of wire segments and programmable switches to connect wire segments. Together, the PLBs and the routing matrix can be connected in different ways to build combinatorial and sequential logic circuits. The routing can also be connected to I/O cells, which interface the FPGA with the other devices via external pins.

All programmable elements of an FPGA are configured using a volatile memory called the configuration memory [5]. Before running an FPGA, the intended design must be downloaded to this configuration memory. The configuration memory is typically arranged in frames tiled across the FPGA [5]. Each frame contains the configuration data for multiple FPGA resources. The number of resources configured by one frame varies from one FPGA to another.

Since the configuration memory reflects the current state of the FPGA resources, most FPGAs also allow reading back the contents of the configuration memory to see the

current state of FPGA resources after an FPGA has been running [5]. This process is known as configuration memory readback.

FPGAs provide a number of configuration interfaces to download a design to the configuration memory or to readback the contents of the configuration memory. Most of these methods use an industry-standard interface such as Boundary Scan (JTAG) to provide serial access to and from the configuration memory [5]. Since a FPGA design is downloaded to the configuration memory as a long stream of data, the configuration downloaded onto an FPGA is known as a bitstream [5].

Modern FPGAs also support partial reconfiguration, where only a portion of the configuration memory is reconfigured while the remaining FPGA device continues its normal operation. Current FPGAs such as the Xilinx Virtex-4 series allow frame-based partial reconfiguration, where a frame is the smallest addressable portion of the configuration memory for download/readback operations [5].

## **1.2 Overview of FPGA Testing**

As the use of FPGAs across various industries increases, so does the need for testing FPGAs for defects and faults. The basic test process involves the application of various test patterns to the FPGA's input, and comparing the output response with that of a known good circuit (or simulation) [3]. If the FPGA-under-test does not return the expected output response, it is deemed faulty. FPGA tests usually require many configurations to test each FPGA resource in different modes of operations. If all test configurations return the expected output response, the device is considered to be fault-free. Testing can be prohibitively expensive for fast and complex systems since the

testing equipment must supply test patterns to the FPGA and collect the output responses faster than the speed of the FPGA itself.

Built-In Self-Test (BIST) is a novel testing procedure that uses the FPGA's own resources to generate test patterns and collect output responses, foregoing the need of expensive external testing equipment [2]. BIST is well-suited for detecting faults in an FPGA without area or performance penalty to the system's intended function [6]. The tester downloads and executes BIST configurations on an FPGA, and reads back the results from the configuration memory to determine if any faults exist in the FPGA. Typically, a tester has to download and execute a set of BIST configurations to exhaustively test all possible faults at each fault location. This provides the maximum fault coverage and diagnostic resolution.

BIST has other advantages beyond the low cost and performance penalties. Since it does not rely on physical access to the FPGA's external pins, the fault coverage is not limited by the number of external pins available on the package [7]. Since the test pattern generators and output response analyzers are on the FPGA itself, BIST can completely test the internal resources of an FPGA regardless of the number of external pins available on the FPGA package.

### **1.3 Verifying BIST Designs using Fault Injection**

BIST has tremendous cost advantages, but it is difficult to experimentally verify the fault coverage and diagnostic resolution of a new BIST for FPGAs without a controlled method of injecting faults into the FPGA [4]. While there has been prior work in fault injection, as discussed in Chapter 2, these methods do not meet the current needs

of the Auburn University Built-In Self-Test (AUBIST) Lab. Based on research objectives, this thesis proposes a fault injection method for Virtex-4 FPGAs where the FPGA is partially reconfigured such that it behaves as a faulty FPGA would. The lab requirements impose several objectives on this fault injection method. First, the method should be able to inject faults in internal FPGA resources. Second, the fault injection needs to be temporary so as not to render the FPGA unusable. Third, the method should utilize the existing Boundary Scan architecture without modifications. Fourth, the method should not place design constraints on the BIST configurations. Fifth, the fault injection should integrate non-intrusively into the overall BIST application process. Finally, the revised BIST application process should be able to verify the BIST algorithm's overall effectiveness for a large list of faults.

This thesis discusses a systematic technique of partially reconfiguring FPGAs such that the behavior of the FPGA emulates an actual fault. The fault emulation is used to inject a fault into the FPGA after each BIST configuration download so that, during the BIST execution, the FPGA behaves in a manner consistent with a faulty FPGA. Since current FPGAs such as Virtex-4 do not support bit-level partial reconfiguration, the proposed technique uses a frame read-modify-write procedure to achieve bit-level partial reconfiguration. It has been verified for known working BIST configurations for FPGAs. A list of acronyms is included in the appendices.

## CHAPTER 2

### BACKGROUND

This chapter provides the background knowledge required to understand fault injection in FPGAs. Since the work described in this thesis was implemented on a Xilinx Virtex-4 FPGA, the chapter begins with a more detailed discussion of the Virtex-4 architecture, configuration memory, and external communication interfaces. The chapter continues with an introduction to FPGA testing with specific emphasis on BIST for FPGAs. This is followed by a discussion of prior work in fault injection for FPGAs. Finally, the chapter concludes with the motivation behind this research, and the underlying objectives that had to be met.

#### **2.1 The Xilinx Virtex-4 FPGA Architecture**

Figure 2.1 shows the basic structure of a Virtex-4 FPGA. The core consists of a two-dimensional array of programmable logic blocks (PLBs), which store eight 4-input Boolean logic functions and eight flip-flop memory elements [8]. The PLBs are interspersed with wire segments and programmable switches called the routing network or the programmable interconnect. By activating various switches in the routing network, the FPGA can string together long lengths of wire segments to connect multiple PLBs with each other. This way, multiple PLBs can be connected to build large combinatorial circuits. Since each PLB also contains flip-flops, they can be wired together to form

large sequential circuits. These circuits interface with the outside world through I/O cells along the boundary of the PLB array.

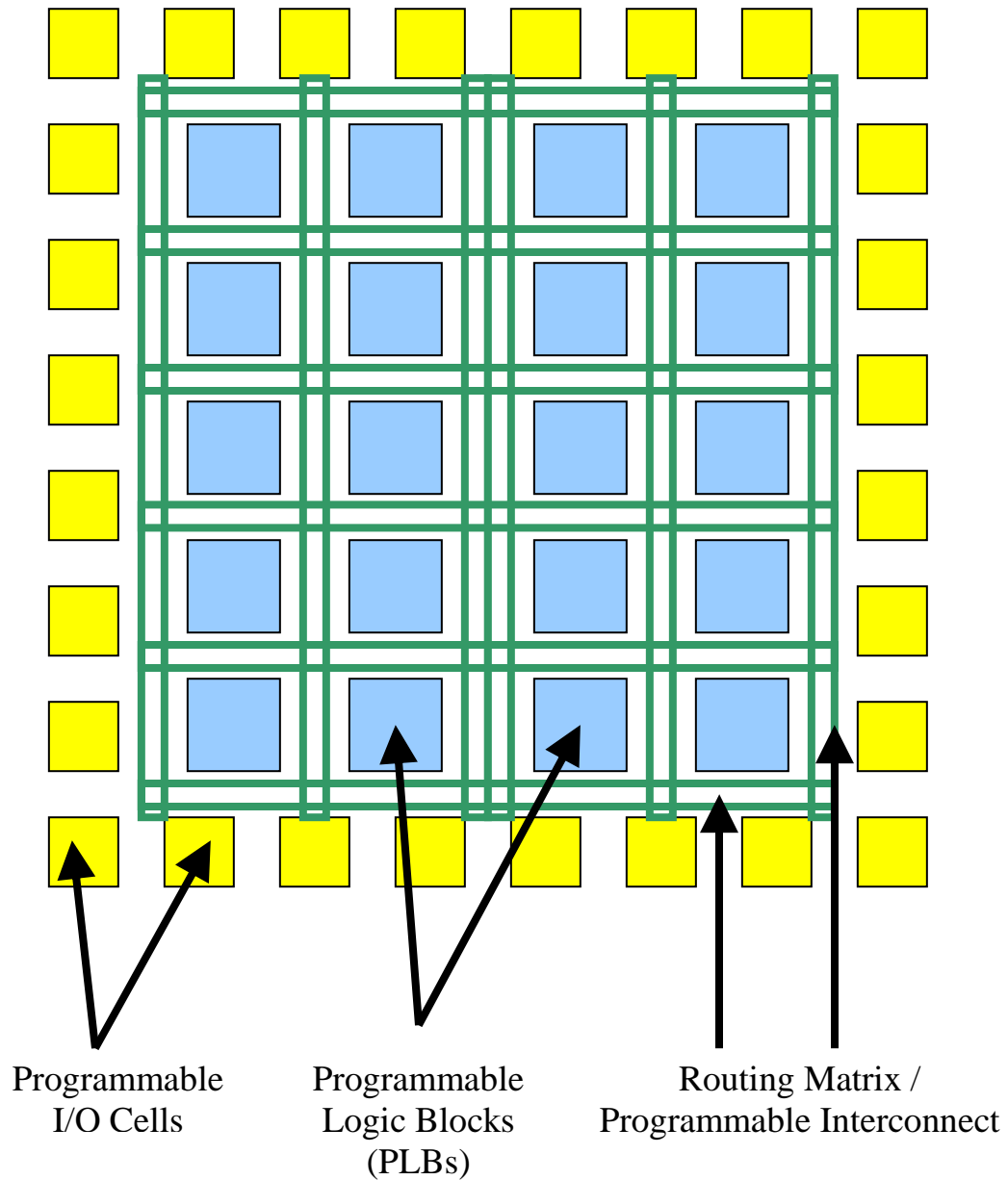


Figure 2.1: FPGA with PLBs, I/O cells, and programmable interconnect

In addition to these typical FPGA features, Virtex-4 includes specialized embedded resources including dedicated RAM, digital signal processors, microprocessors

and high-speed I/O interfaces. The Virtex-4 family of FPGAs is sub-divided into three classes, based on the ratio of these specialized resources.

1. LX for high number of logic resources (PLBs)
2. SX for higher number of DSP resources
3. FX for embedded systems (embedded processor and high-speed I/O interface)

The PLBs in a Virtex-4 are the primarily logic resources of the FPGA to build combinatorial and sequential circuits [8]. As shown in Figure 2.2, each PLB contains four slices, where each slice is connected to the general routing matrix. Each slice contains two look-up tables (LUT-F and LUT-G), which act as four-input Boolean function generators, and two flip-flops, which serve as storage elements. Thus, each PLB contains eight LUTs and eight flip-flops. There are also some fast carry and other common circuitry between LUTs and flip-flops for fast computation.

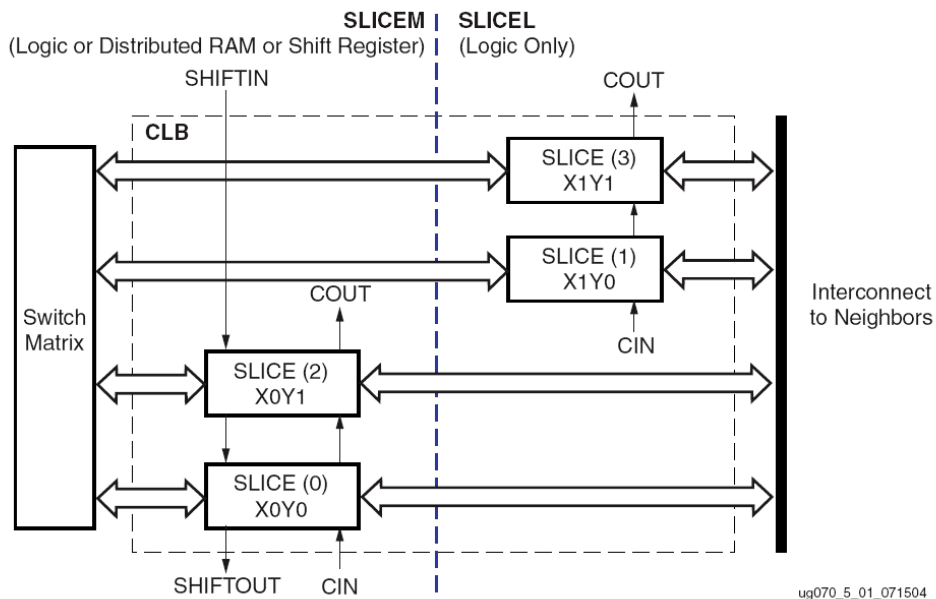


Figure 2.2: One PLB with four slices [8]

Each flip-flop in a slice serves as a storage element. It is configurable as either an edge-triggered D flip-flop or a level-sensitive latch [8]. Each LUT in a slice can describe a 4-input Boolean function [8]. In other words, it is like a truth table with four inputs and one output. Such a truth table can have  $2^4 = 16$  possible input combinations, and one Boolean output value for each input combination, thus a PLB requires 16 bits of configuration information to completely describe its functionality. The LUTs in a slice can also be configured as a shift register or distributed random access memory (RAM).

The routing matrix, also called the programmable interconnect, comprises of wire segments and switches that can be programmed to connect various PLBs. Together, programmable interconnects and PLBs can be configured to build almost any combinational or sequential circuits. Faulty programmable interconnects can have stuck-opens and shorts [1].

I/O cells provide communication between the FPGA's internal resources and the outside components in the system [9]. I/O cells consist of bidirectional buffers and programmable logic such as flip-flops/latches and multiplexers.

Virtex-4 also includes dedicated two-port Block RAMs, where each RAM stores 18Kbits of data [8]. Each port is configurable to treat the block of memory as 16Kx1, 8Kx2, all the way to 512x36. The contents of the block RAM can be defined or cleared by the configuration bitstream.

## 2.2 Virtex-4 Configuration Memory Structure

LUT equations, routing switches, I/O cells and all other programmable resources on a Virtex-4 are configured using a volatile memory called the configuration memory [5]. Before running an FPGA, the intended application design must be downloaded to this configuration memory via a configuration interface. Once the design is downloaded, a startup sequence is performed to initialize the programmable resources and run the FPGA. This complete process is also known as programming, downloading, or configuration of the FPGA.

The configuration memory is arranged in frames tiled across the Virtex-4 FPGA [8], where each frame is a fixed length of 1,312 bits or forty-one 32-bit words. A frame contains the configuration data for 16 PLBs, 4 block RAMs, 32 I/O buffers or 4 digital signal processors [5]. It is the smallest addressable segment of the configuration memory, therefore all memory read/write operations need to be performed on whole frames [5]. This means that individual FPGA resources cannot be reconfigured without also providing explicit reconfiguration data for other FPGA resources that occupy the same frame.

There are no separate address and data buses to access the FPGA's configuration memory. Instead, Virtex-4 provides several configuration registers to accomplish the same task [5]. Frame Address Register (FAR) stores the configuration memory address to where frame data needs to be written or from where frame data needs to be read back. Frame Data Register Input (FDRI) Register is the data input to the configuration memory. Frame Data Register Output (FDRO) Register is the data output from the configuration

memory. There are other registers such as Status (STAT) Register and CRC Register. Given all these registers, all configuration memory operations can be performed by reading and writing to these registers. Therefore, Virtex-4 provides a Command (CMD) Register, which stores the next register operation to perform, such as “Write to FAR” or “Read from FDRI.” By writing a series of such instructions to the CMD register, a configuration download can be performed by initializing the device, writing the frame address to FAR, and writing the configuration data to FDRI. The CMD Register also holds other instructions such as the FPGA startup sequence, which is performed after all configuration data has been downloaded to the configuration memory.

The contents of a configuration memory, starting from frame address 0 onward, make up the configuration bitstream. Instead of providing data for the entire configuration memory, some bitstreams only specify the configuration data for one frame or any number less than the total number of frames. These bitstreams are used to partially reconfigure a portion of Virtex-4 while any existing design downloaded to the configuration memory stays the same. Reconfiguration of only a part of the FPGA is known as partial reconfiguration.

### **2.3 Boundary Scan Interface to the Configuration Memory**

Virtex-4 provides special pins to access the configuration memory from outside the FPGA. These are typically used to configure the device in one of the following modes [5]:

1. Master-serial configuration mode
2. Slave-serial configuration mode

3. Master SelectMAP (parallel) configuration mode
4. Slave SelectMAP (parallel) configuration mode.

However, Virtex-4 also provides the multiple-purpose JTAG configuration interface to the configuration memory. When configured in the proper mode, JTAG's Boundary Scan interface and Test Access Port (TAP) state machine can be used to both download to and read back from the configuration memory [5].

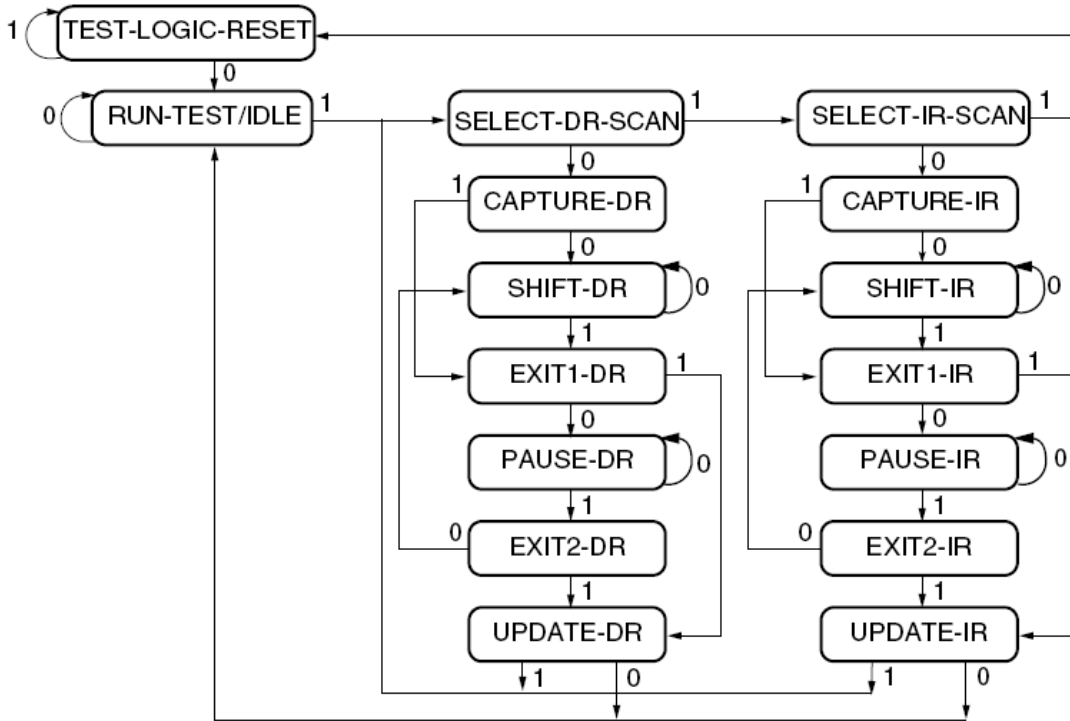
The Boundary Scan is a standard serial communication interface consisting of three input pins to the FPGA and one output pin from the FPGA. The Boundary Scan pins are summarized in Table 2.1.

Table 2.1: Boundary Scan Pin Description

<b>Boundary Scan Pin</b>	<b>Description</b>
TDI	The serial Test Data Input to the FPGA. Configuration memory instructions and data are sent via this pin.
TDO	The serial Test Data Output from the FPGA. Configuration memory contents are read back via this pin.
TMS	The Test Mode Select. Used to cycle through the TAP Controller state machine explained below.
TCK	The Test Clock for cycling through the TAP Controller and shifting data in/out of the FPGA.

Since Boundary Scan only has one control pin (TMS), Virtex-4 provides a TAP Controller state machine (Figure 2.3) to cycle through various configuration and testing modes [5]. Each time the TAP Controller encounters a rising-edge on the TCK, it selects a new state determined by the current TAP state and the logic value on TMS. On the

figure below, the value alongside each state transition represents the logic value on TMS required to make the transition to the designed next state.



NOTE: The value shown adjacent to each state transition in this figure represents the signal present at TMS at the time of a rising edge at TCK.

Figure 2.3: 16-state TAP Controller from Virtex-4 configuration guide

On the figure, the right-most column of TAP states (suffixed “IR”) is used to serially send (“shift in”) Boundary Scan instructions to the FPGA. The two Boundary Scan instructions of our primary interest are CFG\_IN and CFG\_OUT, which respectively provide serial input and output access to the FPGA’s configuration memory. Once a CFG\_IN or CFG\_OUT mode is selected, the middle column of TAP states (“DR”) is used to shift data in/out of the FPGA configuration memory. The “idle” state for TAP is the Run-Test/Idle (RTI) state. Like most Boundary Scan operations, CFG\_IN and

CFG\_OUT operations begin in the RTI state, load the instruction in Shift-IR state, perform data communication in Shift-DR state, and finally return to RTI state.

Table 2.2 shows the sequence of Boundary Scan operations required to enter the CFG\_IN or CFG\_OUT state [5]. Successive chapters of the thesis will refer to loading the Boundary Scan CFG\_IN or CFG\_OUT instruction, which requires the following steps to be performed on Boundary Scan. An understanding of Boundary Scan is important as it is used for both FPGA testing and fault injection.

Table 2.2: Boundary Scan operations to access configuration memory interface

Step	Description	TDI	TMS	# of TCK clocks
1	Begin in RTI state			
2	Move to Select-IR state	X	1	2
3	Move to Shift-IR state	X	0	2
4	Load first 9 bits of instruction (LSB 1st) CFG_IN CFG_OUT	111000101 111000100	0	9
5	Load last bit (MSB) of instruction, while leaving Shift-IR state	1	1	1
6	Move to Select-DR state	X	1	2
7	Move to Shift-DR state	X	0	2
8	Shift data to the configuration memory circuitry (CFG_IN mode) or out of the configuration interface circuitry (CFG_OUT mode), MSB first	X	0	X
9	Write/read last bit from configuration memory while idling out of Shift-DR state	X	1	1
10	Move to TLR state	X	1	5

## 2.4 Built-In Self-Test for Virtex-4 FPGA

Thorough testing of FPGAs is important during both manufacturing and in-system verification. Traditional testing techniques use external automated test equipment (ATE) [3]. The ATE's test pattern generator (TPG) provides the FPGA with a series of test patterns, while the ATE's output response analyzer (ORA) analyzes the FPGA's output response to see if any mismatches occurred compared to the expected output response. Since most ATEs connect to the FPGA's I/O pins, they are not useful for system-level testing. Other ATEs use the serial Boundary Scan interface, but this process is prohibitively slow for testing all internal resources of an FPGA. Regardless, the ATE's clock speed needs to be faster than the FPGA so that it can pass test patterns and collect output responses without missing data. Such high-speed ATEs increase testing costs.

Built-In Self-Test (BIST) for FPGAs is a more viable alternative that is suitable for both manufacturing and system-level testing [2]. In BIST for FPGAs, the TPGs, the resources under test, and the ORAs are all included in the FPGA configuration. Multiple BIST configurations are required to exhaustively test all fault locations and fault types. For each BIST configuration, an external controller downloads the configuration, provides the FPGA with the necessary clock cycles to generate test patterns and produce output responses, and then reads back the contents of the ORAs after each BIST configuration to determine if any faults were detected. Since test patterns and output responses are generated within the FPGA, the test itself is extremely fast. Most of the latency occurs during the download of BIST configurations to the FPGA, and read back of ORA contents from the FPGA.

Once a BIST is complete, the BIST configurations are overwritten with the FPGA's intended system function [2]. In this manner, BIST does not impose any area or performance penalty for the FPGA's intended system function.

There has been significant work in the area of BIST for FPGAs targeting the ORCA 2C series, the Xilinx 4000 series, and the Xilinx Virtex and Spartan series [6][2][1][9]. Current work at the AUBIST lab primarily targets the Xilinx Virtex-4 series. These include BIST for Virtex-4 logic, routing, I/O buffers, RAMs and DSPs.

## **2.5 Prior Work in Fault Injection of FPGAs**

For new BIST techniques that are under development, there is a need for experimental verification of fault coverage and diagnostic resolution. However, experimental verification is difficult without a controlled method of injecting hardware faults before running a BIST sequence [4]. Fault injection for FPGAs has previously been described in [4] and [7], but both approaches are insufficient for current fault injection needs with Virtex-4.

The first published method is the Fault Injection Emulator for Field Programmable Gate Arrays described in [4]. The paper describes a fault injection method where the BIST configuration bitstream is modified right before download so that the FPGA emulates a fault. While the direct-injection of faults into the bitstream minimizes clock cycles required to perform fault injection, this method is not perfect for current needs. For example, this implementation was limited to include only a few stuck-open and stuck-closed faults in programmable interconnects, and a few stuck-at faults in flip-flops and LUTs. The limited set of faults was acceptable for demonstration

purposes, but a thorough analysis of a BIST's fault coverage requires fault injection capabilities for a large list of faults. Moreover, this fault injection method was implemented using the smaller and older ORCA 2C and Xilinx 4000 series FPGAs. The bitstream for these FPGAs was relatively straightforward, where the contents of specific PLBs could be reliably modified by changing the correct bit in the bitstream. This is not as simple with Virtex-4 FPGAs, where PLB mapping is especially difficult when using compression or partial reconfiguration options for BIST configurations. Compression and partial reconfiguration are important features for reducing BIST configuration download times in the much larger Virtex-4 devices. Instead, a post-download fault injection method could work with compressed and partial reconfiguration, and can even be extended to embedded processor-based fault injection.

A post-download fault injection approach for FPGAs has been suggested in [7]. Unlike [4], this approach injects faults after a design has been downloaded to the configuration memory. It uses the existing Boundary Scan architecture in conjunction with user-defined registers implemented on the FPGA fabric to emulate faults on the external pins of the FPGA. This fault injection method may be useful for testing fault tolerance of in-service hardware, but it is not well-suited for verification of BIST designs for several reasons. First, this approach only injects faults into the external pins of an FPGA, which is insufficient for a BIST of internal resources. Moreover, this approach's use of user-defined registers invokes area penalties and design constraints on BIST configurations. An optimal fault injection method would work independent of the BIST configuration under test.

## 2.6 Fault Injection for Verification of Virtex-4 BIST Design

The shortcomings of existing fault injection methods directly lead to a set of requirements for a new fault injection method for Virtex-4 FPGAs. Like the Slaughter method, the Virtex-4 fault injection method should be able to inject faults in internal FPGA resources. Second, the method should utilize the existing Boundary Scan architecture without modifications. Third, the method should not place design constraints on the BIST configuration. Finally, the fault injection should non-intrusively integrate into the BIST process, so that new BIST algorithms can be experimentally verified for a large list of faults.

Faults in LUTs include stuck-at-1 and stuck-at-0 faults. Since PLBs are configured using the configuration memory, these stuck-at faults can be emulated by reconfiguring the FPGA. Since the programmable interconnect is also configured using the configuration memory, stuck-open and shorts can be emulated by reconfiguring the FPGA. Furthermore, a stuck-at-0 or stuck-at-1 fault in any configuration bit can be emulated.

This thesis discusses a fault injection method for Virtex-4 FPGAs, whose goal is to evaluate fault coverage and verify diagnostic resolution of new BIST algorithms. Chapter 3 describes the emulation of a fault in the FPGA by altering the relevant FPGA resource's behavior in the configuration memory. Since current FPGAs such as Virtex-4 do not support reconfiguration of individual FPGA resources, the proposed technique uses a frame read-modify-write procedure to achieve fault emulation.

Chapter 4 discusses the integration of fault emulation as part of the BIST process. By injecting the fault emulation into the FPGA after each BIST configuration download, and then providing the BIST clock cycles, the FPGA runs the BIST while behaving like a faulty FPGA. The BIST results are collected and analyzed to determine the BIST algorithm's fault coverage. For the experimental results of this thesis, the fault injection was tested with known-working BIST configurations for Virtex-4 FPGAs.

## CHAPTER 3

### EMULATION OF FAULT USING PARTIAL RECONFIGURATION

Emulated fault injection for BIST requires the reconfiguration of a single configuration memory bit after each BIST configuration download. However, Virtex-4 does not support the reconfiguration of a single bit. The smallest addressable portion of a Virtex-4 configuration memory is a frame, therefore, complete frames must be written during reconfiguration. This frame-level partial reconfiguration is insufficient for fault injection since each frame contains configuration for several FPGA resources. This chapter discusses a Frame Read-Modify-Write (Frame RMW) technique that uses the built-in frame-based partial reconfiguration and readback features of the FPGA to implement bit-level partial reconfiguration.

Frame RMW can be used to overwrite just a single bit in the configuration memory while keeping the remaining BIST configuration intact. The basic steps to Frame RMW are as follows:

1. Determine the address of the frame that contains data to be modified.
2. Read the frame data from the configuration memory
3. Make the necessary bit changes to the frame data
4. Write the modified frame back to the configuration memory

5. (Optional) Perform a final partial readback to verify Frame RMW results

Sections 3.1 through 3.4 describe the five steps of emulating faults using Frame RMW. Sections 3.5 and 3.6 discuss actual implementation and experimentally show that Frame RMW can reliably change the behavior of an FPGA.

### 3.1 Mapping Fault Location to the Configuration Memory Bit

Since programmable FPGA resources are configured using the configuration memory, a fault location in the FPGA maps to a specific bit in the configuration memory. Virtex-4 has a straightforward convention to determine which frame holds the configuration memory bit that maps to a given fault emulation location. The frame address is a function of the frame's orientation (top or bottom) in the configuration memory, block type, row index, column index and minor address, as shown in Table 3.1.

Table 3.1: A frame's address is based on its physical location on the FPGA

Bit Index	Type	Description
22	Orientation	Top or bottom half of configuration memory
21-19	Block type	PLB/IO/CLK (000), Block RAM interconnect (001), Block RAM content (010)
18-14	Row index	A row of frames, where row 0 is nearest to the center and row index increases away from the center
13-6	Column index	A major column such as column of PLBs. Column 0 is on the left and column index increases rightward.
5-0	Minor index	Selects a specific frame within the major column.

Different FPGA resources require different numbers of configuration bits to define them, and are mapped differently throughout the FPGA. The experimental results

in this thesis are performed using specific faults. Complete mapping of which bit within the frame corresponds to a given FPGA resource is protected under a non-disclosure agreement, but a user may determine the mapping of specific FPGA resources by generating bitstreams using the logic allocation option (-l) with the bitgen.exe software for Virtex-4.

### **3.2 Partial Frame Readback**

Once the address of a frame is determined, the frame data can be read back from the Virtex-4 configuration memory using Boundary Scan. Loading the Boundary Scan instruction CFG\_IN allows the tester to send partial configuration memory readback commands through Shift-DR mode. Similarly, loading the Boundary Scan instruction CFG\_OUT allows the tester to receive frame data through Shift-DR mode.

The series of Boundary Scan and configuration memory commands required for partial readback are described in the Virtex-4 Configuration Guide (page 106) [5]. The steps are summarized as follows:

1. Load Boundary Scan CFG\_IN instruction
2. Go to Boundary Scan Shift-DR state
  - a. Send configuration memory command RCRC
3. Load Boundary Scan JShutdown instruction
4. Go to RTI state and clock 12 times
5. Load Boundary Scan CFG\_IN instruction
6. Go to Boundary Scan Shift-DR state

- a. Send configuration memory command RCRC
  - b. Send configuration memory partial readback commands
7. Load Boundary Scan CFG\_OUT instruction
  8. Go to Boundary Scan Shift-DR state
    - a. Read back frame data from the configuration memory
    - b. Read back pad frame from the configuration memory
  9. Return to Boundary Scan Idle state

**Table 3.2** shows the precise Boundary Scan signals and configuration memory commands required to perform this procedure. The shaded Step 34 is the actual frame readback.

Table 3.2: Boundary Scan signals to load the Boundary Scan CFG\_IN instruction

Step	Description	TDI	TMS	# of TCK clocks
1	Begin in RTI state			
2	Move to Select-IR state	X	1	2
3	Move to Shift-IR state	X	0	2
4	Load first 9 bits of CFG_IN (LSB 1 <sup>st</sup> )	111000101	0	9
5	Load last bit (MSB) of instruction, while leaving Shift-IR state	1	1	1
6	Move to Select-DR state	X	1	2
7	Move to Shift-DR state	X	0	2
8	Shift data to the configuration memory circuitry (MSB 1 <sup>st</sup> )	See below	0	See below

Step	Description	TDI	TMS	# of TCK clocks
	Dummy word	0xFFFFFFFF		32
	Sync word	0xAA995566		32
	No Operation	0x20000000		32
	Write to CMD register	0x30008001		32
	RCRC	0x00000007		32
	No Operation	0x20000000		32
	No Operation	0x20000000		32
9	Write last bit to configuration memory while exiting Shift-DR	X	1	1
10	Move to TLR state	X	1	5
11	Move to RTI state	X	0	1
12	Move to Select-IR state	X	1	2
13	Move to Shift-IR state	X	0	2
14	Load first 9 bits of JShutdown (LSB 1 <sup>st</sup> )	111001101	0	9
15	Load last bit (MSB) of instruction, while leaving Shift-IR state	1	1	1
16	Move to Update-IR state	X	1	1
17	Move to RTI and state for 12 TCK	X	0	12
18	Move to Select-IR state	X	1	2
19	Move to Shift-IR state	X	0	2
20	Load first 9 bits of CFG_IN (LSB 1 <sup>st</sup> )	111000101	0	9
21	Load last bit (MSB) of instruction, while leaving Shift-IR state	1	1	1

Step	Description	TDI	TMS	# of TCK clocks
22	Move to Select-DR state	X	1	2
23	Move to Shift-DR state	X	0	2
24	Shift data to the configuration memory circuitry (MSB 1 <sup>st</sup> )	See below	0	See below
	Dummy word	0xFFFFFFFF		32
	Sync word	0xAA995566		32
	No Operation	0x20000000		32
	Write to CMD register	0x30008001		32
	RCRC	0x00000007		32
	Write to FAR register	0x30002001		32
	Frame address value	X		32
	FDRO Type 1 Read	0x28006000		32
	Read 82 words (41 word frame plus 41-word pad frame) calculated as (0x48000000 + numberOfFrames*41)	0x48000052		32
	No Operation	0x20000000		32
No Operation	0x20000000	32		
25	Write last bit to configuration memory while exiting Shift-DR	X	1	1
26	Move to TLR state	X	1	5
27	Move to RTI state	X	0	1
28	Move to Select-IR state	X	1	2
29	Move to Shift-IR state	X	0	2
30	Load first 9 bits of CFG_OUT (LSB 1 <sup>st</sup> )	111000100	0	9

Step	Description	TDI	TMS	# of TCK clocks
31	Load last bit (MSB) of instruction, while leaving Shift-IR state	1	1	1
32	Move to Select-DR state	X	1	2
33	Move to Shift-DR state	X	0	2
34	<b>Read back frame data</b>	<b>X</b>	<b>0</b>	<b>1,312</b>
35	Read back pad frame	X	0	1,312
36	Read last bit from configuration memory while exiting Shift-DR	X	1	1
37	Move to TLR state	X	1	5
38	Move to RTI state	X	0	1

As the frame is read back from the FPGA configuration memory, it can be stored in an external memory device for bit-level manipulation. The most straightforward method of storing the frame data in external memory is to use a 32-bit word-addressable memory that can hold at least 41 words. Memory address location 0, in this case, corresponds to word 0 of the frame, and so forth.

The frame data retrieved from the configuration memory follows an MSB First order, so the very first frame bit received after the pad frame corresponds to bit 31 of word 0 of the frame. The second bit received is bit 30 of word 0 of the frame, and so forth.

### 3.3 Frame Data Modification

After a frame's data has been read back from the FPGA configuration memory, it can be stored and manipulated in an external memory. Once the required manipulations

have been made to emulate a fault, the memory contents are written back to the configuration memory as explain in the next section.

Single stuck-at faults are emulated using bit-wise operations on the memory. A stuck-at-0 fault is emulated by writing a 0 to the designated memory location determined in Section 3.1. Similarly, a stuck-at-1 fault is emulated by writing a 1 to the designated memory location. If the memory contents are treated as 32-bit words, the fault can be injected using a mask. A specific implementation is described in Chapter 4.

### **3.4 Partial Frame Reconfiguration**

Once the frame data has been modified in the external memory device, it can be written back to the FPGA using partial reconfiguration. Virtex-4's partial reconfiguration feature allows frame data to be written to the configuration memory through Boundary Scan's CFG\_IN instruction. Once the CFG\_IN instruction is loaded, the necessary frame-write commands and the new frame data can be shifted to the configuration memory in Shift-DR mode.

The 32-bit configuration memory commands are sent to the configuration interface circuitry in MSB first order [5]. This set of configuration memory commands can also be reproduced when generating a partial bitstream using BitGen with the CRC:Disable option set. The commands in gray may be skipped to perform an active reconfiguration.

The steps are summarized as follows:

1. Load Boundary Scan CFG\_IN instruction

2. Go to Boundary Scan Shift-DR state
  - a. Send configuration memory command RCRC
  - b. Send Device ID code
  - c. Send COR register value
  - d. Send configuration memory command Shutdown
  - e. Send configuration memory partial reconfiguration commands
  - f. Write frame data
  - g. Write pad frame
  - h. Send configuration memory Startup sequence commands
3. Return to Boundary Scan Idle state

Table 3.3 shows the precise Boundary Scan signals and configuration memory commands required to perform a partial reconfiguration. This Boundary Scan and configuration memory sequence injects the fault using partial reconfiguration and performs a startup on the FPGA.

Table 3.3: Boundary Scan operations for partial reconfiguration

<b>Step</b>	<b>Description</b>	<b>TDI</b>	<b>TMS</b>	<b># of TCK clocks</b>
1	Begin in RTI state			
2	Move to Select-IR state	X	1	2
3	Move to Shift-IR state	X	0	2

Step	Description	TDI	TMS	# of TCK clocks
4	Load first 9 bits of CFG_IN (LSB 1 <sup>st</sup> )	111000101	0	9
5	Load last bit (MSB) of instruction, while leaving Shift-IR state	1	1	1
6	Move to Select-DR state	X	1	2
7	Move to Shift-DR state	X	0	2
8	Shift data to the configuration memory circuitry (MSB 1 <sup>st</sup> )	See below	0	See below
	Dummy Word	0xFFFFFFFF		32
	Sync Word	0xAA995566		32
	No Operation	0x20000000		32
	Write to CMD register	0x30008001		32
	RCRC	0x00000007		32
	No Operation	0x20000000		32
	No Operation	0x20000000		32
	Write to ID register	0x30018001		32
	See appendix for the appropriate sub-device ID code for Virtex-4 devices.	0x_____		32
	Write to COR register	0x30012001		32
	See V-4 Configuration Guide page 93 for options	0x100431E5		32
	Write to CMD register	0x30008001		32
	Shutdown	0x0000000B		32
	No Operation	0x20000000		32
	Write to CRC register	0x30000001		32

Step	Description	TDI	TMS	# of TCK clocks
8	CRC-disable indicator generated by bitgen	0x0000DEFC		32
	No Operation	0x20000000		32
	No Operation	0x20000000		32
	No Operation	0x20000000		32
	No Operation	0x20000000		32
	Write to CMD register	0x30008001		32
	AGHIGH	0x00000008		32
	No Operation	0x20000000		32
	Write to CMD register	0x30008001		32
	WCFG (Write to config. memory)	0x00000001		32
	No Operation	0x20000000		32
	Write to FAR register	0x30002001		32
	The frame address to reconfigure	X		32
	No Operation	0x20000000		32
	Number of words to write: 1 frame + 1 pad frame = 41 words + 41 words = 82 = 0x52	0x30004052		32
	Shift in 41 words of frame data in the same order as RBT/BIT files: Shift 32-bits at a time in MSB-First order.	X		1,312
	Shift in 41 words of pad frame	0		1,312
	Write to CMD register	0x30008001		32
	GRESTORE command	0x0000000A		32
	No Operation	0x20000000		32

Step	Description	TDI	TMS	# of TCK clocks
	Write to CMD register	0x30008001		32
	LFRM command	0x00000003		32
	Shift 101 No Operations	0x20000000		3,232
	Write to CMD register	0x30008001		32
	GRESTORE	0x0000000A		32
	No Operation	0x20000000		32
	Write to CMD register	0x30008001		32
	START command	0x00000005		32
	No Operation	0x20000000		32
	Write to CRC register	0x30000001		32
	CRC-disable value generated by bitgen	0x0000DEFC		32
	Write to CMD register	0x30008001		32
		0x0000000D		32
	No Operation	0x20000000		32
	No Operation	0x20000000		32
	No Operation	0x20000000		32
	No Operation	0x20000000		32
9	Write last bit from configuration memory while exiting Shift-DR	X	1	1
10	Move to TLR state	X	1	5
11	Move to RTI state	X	0	1

The sequence of partial configuration memory readback, frame modification and partial reconfiguration causes the FPGA to emulate the behavior of a fault. This “fault injection” can be integrated into a BIST procedure to determine if the BIST can accurately detect the fault. Since the procedure can be performed by an automated test program, a BIST’s performance can be determined by subjecting it to a large list of faults. Chapter 4 discusses the integration of the fault injection step into the BIST sequence.

### **3.5 Implementation of Fault Emulation in C**

Fault injection commands can be sent to the FPGA using any external processor using Boundary Scan. This implementation uses a C program running on a Windows PC since other BIST development efforts in the AUBIST lab use the same platform. Figure 3.1 shows a basic implementation setup, where the Virtex-4 device is connected to a standard Windows PC using the Xilinx JTAG Parallel Cable IV running in “Parallel Cable III” (PC-III) mode. The PC-III mode maps Boundary Scan pins on the FPGA to parallel port pins on the PC. Consequently, a C program running on the PC can control the FPGA’s Boundary Scan by reading and writing to parallel port pins. The TCK pin is clocked by writing a 1 followed by a 0. During the TCK’s inactive state (when a 1 has been written to TCK but before a 0 is written), the program can change TMS, TDI and TDO values. By changing the TMS values, the C program can cycle through the various TAP states defined in Section 2.3. Once in the proper TAP state, the C program can communicate with the configuration memory interface circuitry using TDI and TDO pins.

Configuration memory commands and new frame data is sent serially through TDI.  
 Configuration memory contents are read back through TDO.

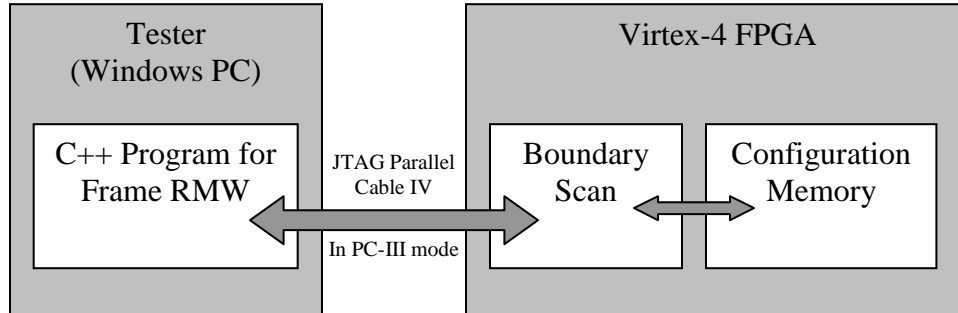


Figure 3.1: Frame RMW performed using Windows-based Application

The Frame RMW procedure for bit-level partial reconfiguration has been implemented in C and tested for LX25, LX60, FX12, and SX35 devices. It has been added on top of an existing Windows-based application developed by the Auburn University Built-In Self-Test (AUBIST) Lab. The existing application, Virtex-4 BIST GUI, provides the low-level functions to initialize the parallel port access to the FPGA, to move between various TAP states, and to shift data in/out of Boundary Scan via TDI/TDO. The Frame RMW implementation uses these predefined functions to perform frame read/write operations on the configuration memory.

For this implementation, the Frame RMW function signature is as follows:

```
modifyFrameWord (unsigned frameAddress, int wordIndex, int
newValue, int mask)
```

- **frameAddress** is the frame to read/modify/write.
- **wordIndex** is the word of the frame to modify, where wordIndex=0 is the first word.
- **newValue** is the new 32-bit value of the word.
- **mask** determines which bits of the word to modify.

The function performs the following memory operation based on the `wordIndex`, `newValue` and mask information:

```
frameData[wordIndex] = (frameData[wordIndex] & (~mask)) +
(newValue & mask);
```

Note that Frame RMW is a generic function that modifies the contents of the configuration memory given a specific frame address, `wordIndex`, new word value, and mask. For single-bit fault injection, a higher-level function is needed to calculate the frame address, word index, new value, and mask, before calling the Frame RMW function. Table 3.4 shows the C implementation of **injectFault**. The frame address is calculated based on the rubric from Section 3.1.

Table 3.4: Fault Injection Function Source Code

```
void CVirtexBISTDtg::injectFault (int tb, int blockType, int row, int
col, int frame, int wordIndex, int bitIndex, int stuckAt){
    int frameAddress, newValue, mask;
    CString strTemp;

    // Prepare frame modify parameters
    frameAddress = ((tb          & 0x1)  << 22) |
                  ((blockType & 0x7)  << 19) |
                  ((row       & 0x1F) << 14) |
                  ((col       & 0xFF) <<  6) |
                  (frame      & 0x3F);

    mask = (1<<bitIndex);           // Create mask
    newValue = (stuckAt<<bitIndex); // Create new value of word

    // Read/Modify/Write Frame
    modifyFrameWord(frameAddress, wordIndex, newValue, mask);
}
```

### 3.6 Experimental Results of Fault Emulation

To determine if the Frame RMW function can accurately change the behavior of an FPGA, it was used to change the LUT equation in a specific PLB in the FPGA. The results were deemed successful if, (a) changing the LUT contents changed the expected behavior of the FPGA, (b) a readback showed that the entire configuration memory's contents were the same except for the specific LUT bits changed using Frame RMW.

To demonstrate the reconfiguration of a single LUT function generator using Frame RMW, a reference FPGA design was generated that contained a simple two-input AND function implemented in LUT G of SLICE\_X54Y2 of an LX25. Two inputs to the LUT connected to onboard switches, while the other two inputs were “don't care”. The output connected to an onboard LED. Downloading this reference design to an LX25 device yielded a basic AND gate operated by the two switches.

The logic was stored in LUT G of SLICE\_X54Y2, whose configuration was stored in frame 0x4087D5, word 37 (i.e. 38<sup>th</sup> word), and bits 9-24 (where bit 0 is LSB of the word). The exact mapping of all LUTs (and other programmable resources) to their locations in the configuration memory is proprietary information, but the explicit mapping provided for this experiment will suffice.

Next, the new contents of the LUT were determined using an XOR truth table for a 4-input LUT, where A and B are the switch inputs while C and D are don't cares. The output of the truth table in Table 3.5, written as a 16-bit binary number, is as follows: 0000111111110000b. The equivalent hexadecimal number would be 0x0FF0.

Table 3.5: 4-Input XOR Truth Table

<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>Z</b>
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Since the new LUT contents 0x0FF0 began at bit 9 and ended at bit 24, the bit locations for the frame word before and after the LUT contents were filled with zeros to yield a new 32-bit word 0x001FE000. Since only bits 9-24 should be changed, the mask value was set to 0x01FFE000:

```
modifyFrameWord(frameAddress, wordIndex, newValue, mask)
```

```
modifyFrameWord(0x4087D5, 37, 0x001FE000, 0x01FFE000);
```

The immediate evidence that the Frame RMW procedure worked was that the FPGA now performed an XOR operation on the two switch settings rather than an AND. This completed the behavioral verification of Frame RMW.

To ensure that no other frame bits were modified, a “verification readback” was performed after Frame RMW. The verification readback frame was compared to the original frame, and no differences were found other than the LUT equation contents. The experiment was repeated for LX60, FX12 and SX35 with the same successful results.

These experiments have shown how Frame RMW can modify any bit in the configuration memory. The higher-level injectFault function can appropriately call Frame RMW to change the FPGA’s behavior to emulate stuck-at-1 or stuck-at-0 faults. A more pertinent experiment to demonstrate successful fault emulation would be the use of this method to inject specific faults in the FPGA, and then checking if a previously published BIST approach can detect those faults. Chapter 4 begins with the integration of this fault injection method into the BIST sequence. Experiments in that chapter involve the injection of a list of faults into the FPGA to see if a previously published BIST approach can accurately detect them.

## CHAPTER 4

### INTEGRATION OF FAULT INJECTION INTO THE BIST PROCEDURE

BIST configurations are downloaded to the FPGA via a configuration interface such as Boundary Scan. The ORA results are read back from the FPGA via the same interface. Since the fault injection method described in Chapter 3 also uses Boundary Scan, the fault injection can be performed by the same test program that also performs the BIST download and readback. In effect, fault injection can be integrated non-intrusively into the BIST process without additional hardware changes. The external processor that performs BIST download, fault injection and readback can also be modified to repeat the process for a large number of faults, and compile the results to determine experimental fault coverage.

This chapter begins with a theoretical overview of integrating fault injection into existing BIST processes. This overview is followed by actual implementation in C. The chapter ends with experimental results of injecting faults and detecting them using Logic BIST.

#### **4.1 Integration Overview**

Table 4.1 shows the standard steps of BIST for FPGAs. Each BIST configuration includes the TPGs and ORAs for testing a sub-set of the total possible faults. Complete

testing for all faults requires the download, running, and read back of results from all  $N$  BIST configurations.

Table 4.1: Original BIST process without fault injection

1. Let  $N$  = Number of BIST configuration
2. For  $I = 1$  to  $N$ 
  - a. Download  $I^{\text{th}}$  BIST configuration to the FPGA
  - b. Provide BIST clock cycles to FPGA
  - c. Retrieve ORAs from the FPGA
3. Save results to file.

Since fault injection is emulated using partial reconfiguration, it is overwritten every time a new BIST configuration is downloaded to the FPGA. Thus, any fault injected after downloading the first BIST configuration must be re-injected following each BIST configuration download. Consequently, Table 4.2 shows a revised BIST process with a fault injection step after each BIST configuration download. Once the BIST process is complete, the results can be analyzed to determine if they detect the same fault that was injected.

Table 4.2: Revised BIST process with additional fault injection step

1. Let  $N$  = Number of BIST configuration
2. For  $I = 1$  to  $N$ 
  - a. Download  $I^{\text{th}}$  BIST configuration to the FPGA
  - a. If fault injection requested**
    - i. Read fault location and value from variables**
    - ii. Inject fault into the FPGA**
  - b. Provide BIST clock cycles to FPGA
  - c. Retrieve ORAs from the FPGA
3. Save results to file.

One of the goals of fault injection is to provide an experimental measure of fault coverage. To this end, the BIST process is further revised to test the FPGA for a large

number of faults. If the test program is provided with a large fault list, Table 4.3 shows how BIST is repeatedly performed for each fault in the list. For each fault, the test program records which BIST configuration(s) detected the fault, and saves this information to a fault results file. The results can be analyzed to determine the fault coverage, as well as which BIST configurations detected a given fault.

Table 4.3: BIST process for a large fault list

1. Let  $M$  = Number of faults in the fault list
2. For  $J = 1$  to  $M$ 
  - d. Get  $J^{\text{th}}$  fault location and value from fault file
  - e. Save fault location and value to variable; set fault injection flag
  - f. Call the revised BIST process in Table 4.2
  - g. Append results to fault results file
3. Save results to file.

## 4.2 Implementation in Virtex-4 BIST GUI

During the BIST development process, it is convenient to use a sophisticated external BIST controller to download BIST configurations, supply BIST clock cycles, and read back the ORAs for analysis. The AUBIST lab has developed a Windows XP-based application (BIST GUI) that serves this purpose. The application holds a repository of BIST configurations for different internal FPGA resources and the three families of Virtex-4 devices. The user can visually specify which Virtex-4 device is connected to the Windows XP system via Boundary Scan. The user can then select from a variety of BIST methods and options before initiating a BIST process. Since the BIST GUI already provides Boundary Scan communication functions, it was a natural extension of the application to implement fault injection.

Figure 4.1 shows the levels of abstraction for implementing BIST with fault injection. The gray cells show the existing GUI functionality. When a user calls the “Run BIST” function, the function performs the BIST process steps outlined in previously outlined in Table 4.1, namely: the function performs configuration memory download, provides the necessary BIST clock cycles, reads back the ORAs, and reports the results to a file.

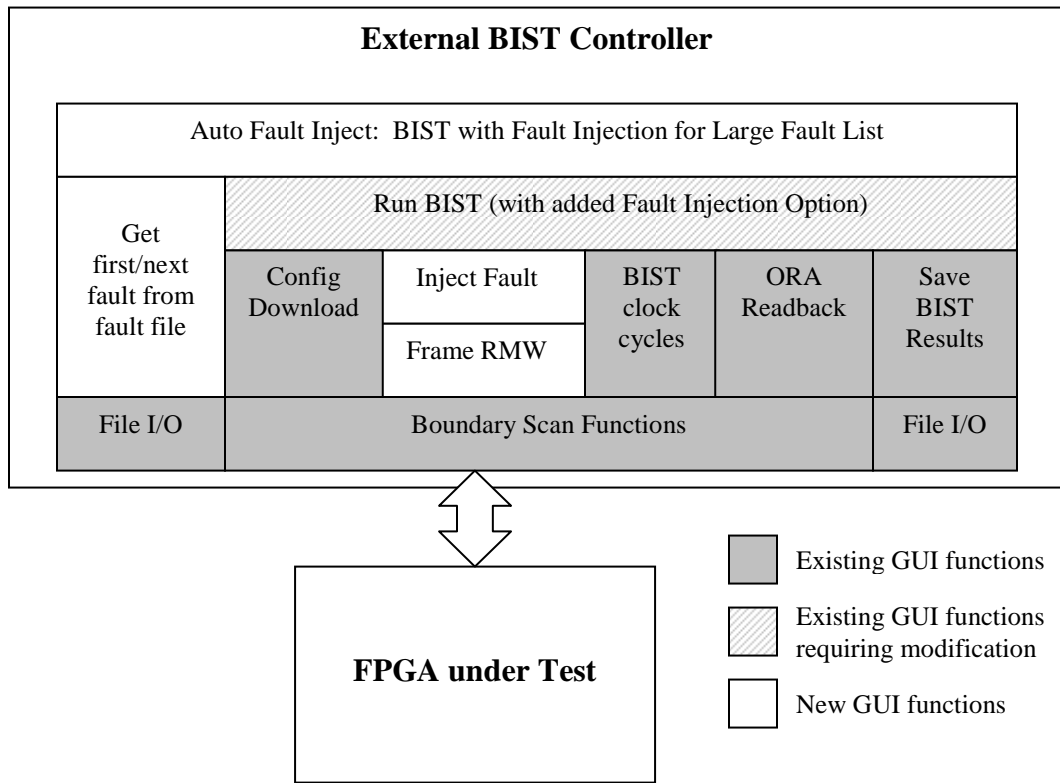


Figure 4.1: Software functions to implement BIST with fault injection

“Run BIST” calls the low level Boundary Scan functions to communicate with the FPGA. The white boxes show the new fault-injection functionality that needs to be added to the program. Specifically, the “Run BIST” function needs to be modified to call an “Inject Fault” function if fault injection is requested by the user. This modification to

the “Run BIST” function, as previously outlined in Table 4.2, is implemented in the following C statement:

```
if (m_fltinject == TRUE)
    injectFault(fltTB,fltBlockType,fltRow,fltCol,fltFrame
    ,fltword,fltBit,fltval);
```

The “Inject Fault” function should perform the Frame RMW processes of frame address determination, frame readback, frame modification and frame reconfiguration discussed in Chapter 3. In the future, Frame RMW may be need by tasks other than fault injection, so the core read-modify-write functionality is implemented at a further low-level function called “ModifyFrameWord,” which uses Frame RMW to change configuration memory contents within frames.

Figure 4.1 also shows a new higher-level function, called “Auto Fault Inject,” which is useful for testing BIST configurations for a large number of faults without manual intervention. “Auto Fault Inject” repeatedly calls the “Get Fault” and “Run BIST” functions to perform fault-injection BIST for a large list of faults. The BIST results for all faults are compiled in a fault results (fltres.txt) file. The specific C implementation is shown in Table 4.4.

Table 4.4: "Auto Fault Inject" function source code

```
// Get next fault
while (getflt() == 0){
    // Open fault results file (in append mode)
    fff = fopen (m_fltres,"a");

    // write fault description to result file
    if (fltTB == 0) fprintf (fff,"Tb%dr%dc%df%dw%db%d sa%d\n",
    fltBlockType,fltRow,fltCol,fltFrame,fltword,fltBit,fltval);
    else fprintf (fff,"Bb%dr%dc%df%dw%db%d sa%d\n",
    fltBlockType,fltRow,fltCol,fltFrame,fltword,fltBit,fltval);
```

```

    // Close file
    fclose (fff);

    // Run BIST processes
    OnrunBIST();
}

```

The implementation for “Get Fault” in C is shown in Table 4.5. First, the function opens the fault file using **fopen** if it has not already been opened. Next, the function uses **fscanf** to get the next fault location and value, and stores the information in global variables. Upon reaching the end of the fault file, the function calls **fclose** to close the fault file.

Table 4.5: "Get Fault from Fault File" function source code

```

int CVirtexBISTDlg::getflt(){
    //int tb, blockType, row, col, frame, wordIndex, bitIndex, fltval;

    // Get first fault from fault list.
    if (fltOpen == 0){
        if ((fpf = fopen(m_fltin,"r")) == NULL){
            sprintf (scan,"can't find fault file '%s'",m_fltin);
            writeLog("Fault", scan);
            return (1);
        }
    }
    fltOpen = 1;
    if (fscanf (fpf,"%s",scan) != EOF){
        // Format: Tb0r0c1f21w1b0
        fltTB = (scan[0]=='T' || scan[0]=='t' ? 0 : 1); //
top/bottom
        sscanf
(&scan[1],"b%dr%dc%df%dw%db%d",&fltBlockType,&fltRow,&fltCol,&fltFrame
,&fltword,&fltBit); //other params

```

```

}
else {
    fclose (fpf);
    fltOpen = 0;
    return (1);
}
if (fscanf (fpf,"%s",scan) != EOF){
    // Stuck-at bit
    sscanf (scan,"%d",&fltval);
    return (0);
}
else {
    fclose (fpf);
    fltOpen = 0;
    return (1);
}
}
}

```

The fault list is stored in an ASCII file. Each fault is separated by a line. The format for designating a fault location and value is as follows:

**Tb0r2c1f21w40b31            1**

The first character can be **T** or **B**, designating the top or bottom half of the FPGA, respectively. The top/bottom designation is followed by **b** and a number, where the number is the block type. The number following **r** is the row number; following **c** is the column number; following **f** is the minor frame number; following **w** is the word index within the frame, starting at zero; and finally, the number following **b** is the bit index within the word, where 0 is the least significant bit. The bit index is following by a tab character (or any whitespace) and then a 1 or a 0. This number corresponds to either a stuck-at-1 or stuck-at-0 fault to be injected and emulated.

The practical integration of fault injection into the BIST process provides a convenient method of experimentally testing accuracy and fault coverage of BIST approaches. Before it can be put to use, however, it needs to be verified whether the fault injection implementation actually emulates the fault that it is intended to emulate. This can be done by creating a sample fault list, and running the BIST GUI with an existing BIST approach to see if the BIST results detect the correct faults.

### **4.3 Experimental Results**

To ensure that the fault injection implementation in the BIST GUI works correctly, it was tested with Logic BIST for Virtex-4 FPGAs described in [2]. A fault list composed of various logic faults was generated. The fault list was supplied to the BIST GUI, which performed a Logic BIST with fault injection for each fault, and collected the results. The results were summarized in a fault results file for later analysis.

When using full and compressed configuration Logic BIST, fault injection has been successfully demonstrated to work for logic resources in devices from each of the three Virtex-4 families – LX60, SX35 and FX12.

When using partial configuration BIST, however, the BIST GUI reported detecting a fault, but was unable to identify which BIST configuration(s) detected the fault. This is because partial configuration BIST did not clear the ORAs between BIST configurations, so once the fault was detected, the ORA read backs showed mismatches even for BIST configurations that did not detect the fault. A possible solution would be to clear the ORAs between successive BIST configurations. Further discussion on this issue is in Section 6.2.1, which discusses areas of future research and development.

## CHAPTER 5

### BOUNDARY SCAN OPERATIONAL TEST

The fault injection method works independent of the FPGA fabric, but relies on the proper operation of the Boundary Scan interface. The need for proper Boundary Scan operation is even more important if fault injection is implemented on an embedded processor that uses Boundary Scan's user-defined registers to communicate with the tester. This chapter describes a test for exhaustively testing a Boundary Scan's pins, user-defined registers, and the User Access Port. The tester first downloads a test configuration to the FPGA via Boundary Scan. The tester then shifts of a series of patterns to the FPGA via Boundary Scan, while simultaneously reading back the results. If the results are as expected, the Boundary Scan interface is deemed operational.

The Boundary Scan Operational Test checks for any stuck-at faults and verifies the proper operation of each signal used by the Boundary Scan USER modules and the User Access Register (UAR). The following Boundary Scan and UAR signals are tested for faults:

- For each Boundary Scan User Module (1 through 4)
  - DRCK (Data Register Clock)
  - TDI (Test Data In)
  - TDO (Test Data Out)
  - SHIFT
  - SEL (1 for selected USER mode; 0 for all others)
  - CAPTURE

- UPDATE
  - RESET
- For the UAR
  - All 32 bits of the UAR register
  - UAR Data Valid flag

The tester first downloads a configuration onto the FPGA containing the Boundary Scan USER modules and related logic. The tester then performs a series of data shifts to/from the FPGA fabric through the four Boundary Scan USER modules and the UAR. The data shifted out of the FPGA is compared with expected results to determine if any faults exist in the Boundary Scan and UAR circuitry.

## **5.1 Overview of Boundary Scan User-Defined Registers**

Boundary Scan provides ways to communicate with the Virtex-4 fabric after the FPGA has been configured. These include the four Boundary Scan USER registers and the 32-bit User Access Register (UAR). The Boundary Scan Operational Test verifies the proper operation of all data and control signals used in these two methods to communicate with the FPGA fabric.

The first method of communicating with the FPGA design is through the use of USER1, USER2, USER3 and USER4 registers. These N-bit registers are defined by the user in the FPGA design. Each user-defined register interfaces with Boundary Scan using one of four available BOUNDARY\_SCAN\_VIRTEX4 modules. The USER register's serial-in, serial-out, shift and clock signals are connected to the BOUNDARY\_SCAN\_VIRTEX4 module's TDI, TDO, SHIFT and DRCK (data register clock) ports, respectively. The DRCK clock is only active for the Boundary Scan module whose associated Boundary Scan instruction is loaded in the Boundary Scan instruction register.

Data can be shifted in and out of these registers by loading the Boundary Scan instructions USER1, USER2, USER3 and USER4, and shifting data in “Data Register Shift” mode (SHIFT-DR). The USER registers also perform data reset, parallel data input, and parallel data output operations by responding to Boundary Scan signals RESET, CAPTURE and UPDATE, respectively. The Boundary Scan/UAR Test checks each of the above-mentioned Boundary Scan signals (DRCK, TDI, TDO, SHIFT, CAPTURE, UPDATE, and RESET) for any faults.

Another method of providing data to the FPGA fabric is by setting the 32-bit UAR in the configuration memory interface circuitry. The user access register contents are available to the FPGA fabric through the user access register module (USR\_ACCESS\_VIRTEX4). It has a 32-bit output port holding the current contents of the UAR. It also has a data\_valid signal that is asserted for one clock cycle (of the configuration clock) whenever new contents are written to the UAR and available to the FPGA fabric. The Boundary Scan Operational Test writes to the UAR, transfers its contents to the FPGA fabric, and then shifts it out via the USER modes to check for any stuck-at faults in the UAR or its Data Valid flag.

## **5.2 Test Configuration**

Figure 5.1 shows the circuit design for the 12-bit user-defined shift register (USER register) connected to one of the four Boundary Scan modules on the FPGA fabric. The design is replicated for each of the four Boundary Scan modules. The TDI and TDO from the Boundary Scan module are respectively connected to the serial-in and serial-out ports of the 12-bit USER register. The SHIFT and DRCK signals from the Boundary

Scan module are also provided to the USER register as shown in the diagram. This establishes the basic serial connection between the tester and the FPGA fabric.

The lower 8 bits (bits 0-7) of the USER register are treated as “user data” while the upper 4 bits (bits 8-11) are used for control. Currently, only bit 11 (PDATAControl) is used to control the source from which the USER register captures its parallel data.

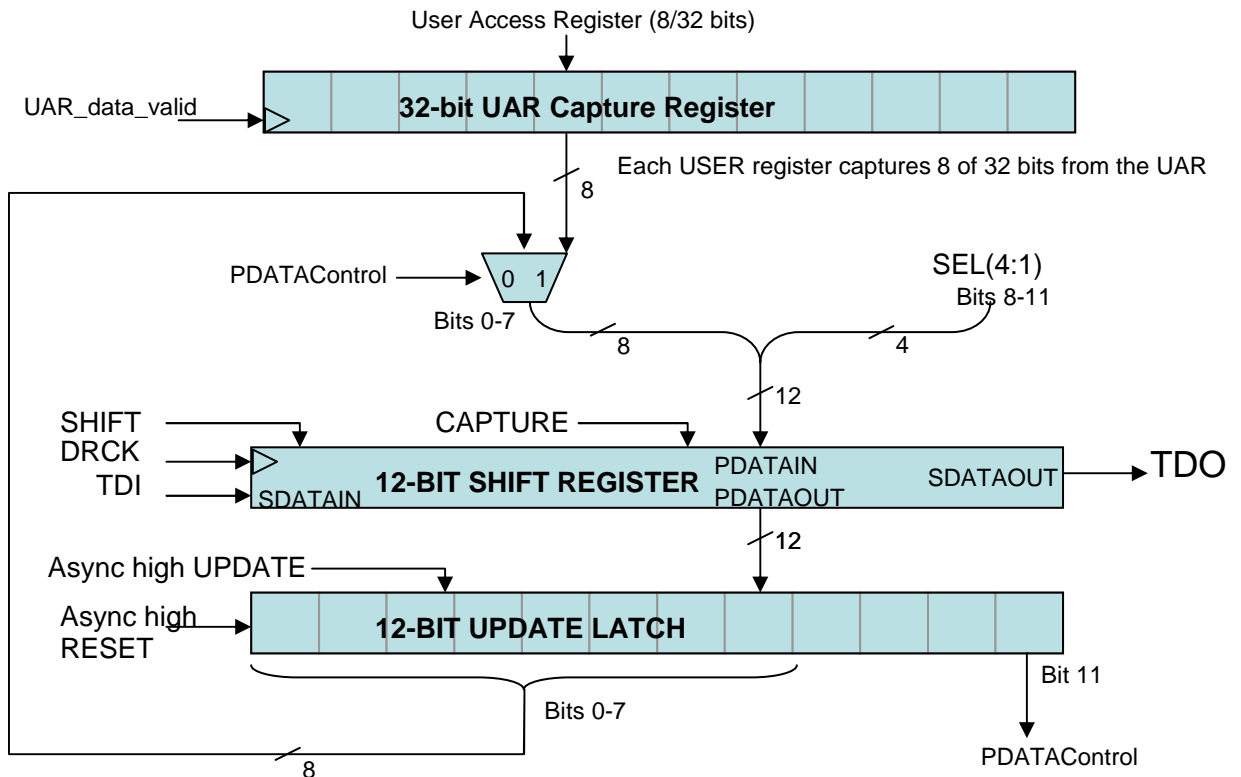


Figure 5.1: USER Register and related test circuitry for Boundary Scan User Module

The shift register also supports 12-bit parallel data input and output. The PDATAOUT holds the 12-bit contents of the USER register at all times. Whenever the Boundary Scan UPDATE signal goes high, the contents of the USER register are transferred into a 12-bit Update Latch as shown in the figure. The lower 8 bits of the latch are routed back to the parallel input PDATAIN of the USER register. These 8 bits are coupled with 4 bits SEL(4:1) that indicate which USER register is currently active; 1

corresponds to an active module while a 0 corresponds to an inactive module. The combined 12-bit bus signal is provided as input to PDATAIN. The USER register captures this 12-bit input whenever the Boundary Scan CAPTURE signal is high during the active edge of DRCK.

In some parts of the test, we'd like to capture the UAR contents into the USER register instead of capturing the latched data back into the USER register. A 2-to-1 multiplexer is used to switch between these two possibilities. The multiplexer is controlled using bit 11 of the Update Latch, as shown in the figure above. Since the UAR is 32-bits, 8 of its 32 bits are captured by each USER register. USER1 captures UAR(7:0). USER2 captures UAR(15:8). USER3 captures UAR(23:16). USER4 captures UAR(31:24).

The figure below shows the internal construction of the USER register. It shows how each flip-flop (FF) of the shift register captures serial or parallel data. If CAPTURE is high, the FF captures the value from PDATAIN. If SHIFT is high, the FF captures the value of the previous FF. These two possibilities are OR'd together to provide the input to the FF. As we will see later, implementing an OR rather than a multiplexer is crucial to detecting faults.

The only "special cases" for these shift register bits are the end bits SR(0) and SR(11). Data serially shifted into the USER register from TDI goes into SR(0). Data shifted out of SR(11) goes to TDO.

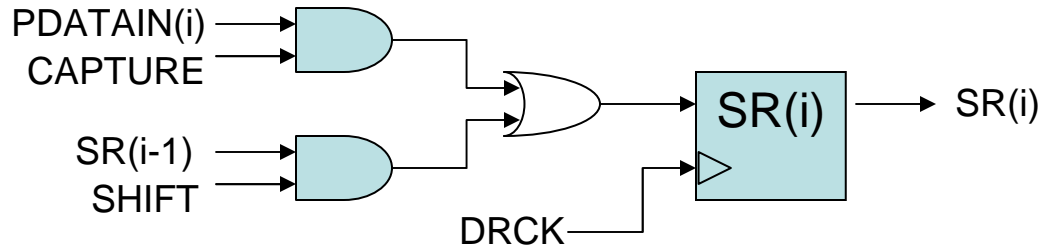


Figure 5.2: Implementation of a single bit SR(i) inside the USER register

Having developed this test configuration, all the Boundary Scan and UAR signals can be tested for faults using the procedure described in the next section.

### 5.3 Test Procedure

The procedure below outlines the Boundary Scan operations performed by the tester to check each Boundary Scan and UAR signal for faults. The procedure is divided into phases 1-4, where the Phase 3 has two sub parts. All changes to TAP state and shifting of data shown below is performed using Boundary Scan pins TCK, TMS, TDI and TDO.

#### PHASE 1

- Begin Boundary Scan in RTI state
- Move TAP state to TLR
- For I = 1 to 4
  - Load Boundary Scan instruction USERI
  - Move to SHIFT-DR state
  - Write TDI (12-bits, MSB first) => 0x0II
  - TDO should read 0xS00
    - (where S is all zeros except (I-1)<sup>th</sup> bit)

#### PHASE 2

- For I = 1 to 4
  - Load Boundary Scan instruction USERI
  - Move to SHIFT-DR state
  - Write TDI (12-bits, MSB first) => Bar(0x0II)
  - Expected TDO result <= 0xSII

- (where S is all zeros except (I-1)<sup>th</sup> bit)

### PHASE 3A

- Write user access register: 0x87654321
- For I = 1 to 4
  - Load Boundary Scan instruction USERI
  - Move to SHIFT-DR state
  - Write TDI (12-bits, MSB first) => 0xFFFF
  - Expected TDO result <= 0xSUU
    - (where S is all zeros except (I-1)<sup>th</sup> bit)
    - UU are 8 bits from UAR.
      - If I=1, UU=UAR(7 downto 0)=0x21
      - If I=2, UU=UAR(15 downto 8)=0x43
      - If I=3, UU=UAR(23 downto 16)=0x65
      - If I=4, UU=UAR(31 downto 24)=0x87

### PHASE 3B

- Write user access register: 0x789abcde (negation of 0x87654321)
- For I = 1 to 4
  - Load Boundary Scan instruction USERI
  - Move to SHIFT-DR state
  - Write TDI (12-bits, MSB first) => 0xFFFF
  - Expected TDO result <= 0xSUU
    - (where S is all zeros except (I-1)<sup>th</sup> bit)
    - UU are 8 bits from UAR.
      - If I=1, UU=UAR(7 downto 0)=0xde
      - If I=2, UU=UAR(15 downto 8)=0xbc
      - If I=3, UU=UAR(23 downto 16)=0x9a
      - If I=4, UU=UAR(31 downto 24)=0x78

### PHASE 4

- For I = 1 to 4
  - Move TAP state to TLR
  - Load Boundary Scan instruction USERI
  - Move to SHIFT-DR state
  - Write TDI (12-bits, MSB first) => 0x0II
  - Expected TDO result <= 0xS00
    - (where S is all zeros except (I-1)<sup>th</sup> bit)

## 5.4 Fault Detection in Boundary Scan USER Registers

This section describes how faults in different portions of the USER modules can lead to mismatched output responses. A comparison between the expected and actual output response helps diagnose the fault.

### 5.4.1 TDI

Phase 1 of the test procedure shifts 12-bit data  $0x0II$  into each of the four USER registers, where  $I$  is the current USER register. For instance,  $0x011$  is shifted into USER1,  $0x022$  into USER2, and so forth.

When exiting the Boundary Scan Shift DR mode for each USER register in Phase 1, the data is latched into a 12-bit Update Latch. When beginning Phase 2, the lower 8 bits of this number are captured back into the USER register. The remaining upper 4 bits of the USER register capture the SEL(4:1) bus signal, which in case of USER1 is '0001' or  $0x1$ , '0010' or  $0x2$  for USER2, '0100' or  $0x4$  for USER3, etc.

As Phase 2 shifts in the new set of 12-bit data into each USER register, the data already in the USER register is shifted out via TDO. In fault-free circuitry, the expected result out of TDO in Phase 2 for USER1 should be  $0x111$ , USER2 should be  $0x222$ , USER3 should be  $0x433$  and USER4 should be  $0x844$ . As stated earlier, the upper 4 bits are SEL(4:1) while the lower 8 bits should be the same as the lower 8 bits of the number shifted-in in Phase 2.

If TDI is stuck-at-0 or stuck-at-1, any data shifted into a USER register in Phase 1 will be stored as all zeros or all ones, respectively. The all-zeroes or all-ones data will be

updated into the Update Latch and its lower 8 bits captured back into the USER register in Phase 2.

For USER1 in Phase 2, an actual readback of 0x100 would indicate TDI is stuck-at-0. An actual readback of 0x1FF would indicate TDI is stuck-at-1.

For USER2 in Phase 2, an actual readback of 0x200 would indicate TDI is stuck-at-0. An actual readback of 0x2FF would indicate TDI is stuck-at-0.

For USER3 in Phase 2, an actual readback of 0x400 would indicate TDI is stuck-at-0. An actual readback of 0x4FF would indicate TDI is stuck-at-0.

For USER4 in Phase 2, an actual readback of 0x800 would indicate TDI is stuck-at-0. An actual readback of 0x8FF would indicate TDI is stuck-at-1.

#### **5.4.2 TDO**

If TDO is stuck-at-0 or stuck-at-1, the readback results will always be all-zeros or all-ones, respectively. If TDO is non-operational for only one Boundary Scan User mode, it may indicate a stuck-at fault in the AND/OR gates that combine TDO1-TDO4.

#### **5.4.3 SHIFT**

To understand the expected results due to stuck-at faults in SHIFT, let's take another look at Figure 5.2. When Boundary Scan is in USER SHIFT-DR mode, SHIFT should be high and CAPTURE should be low. If SHIFT is stuck-at-0, the input to SR(i) flip flop will always be zero, resulting in all USER register contents clearing out. Therefore, if TDO always returns zeros, it may indicate a SHIFT stuck-at-0.

If SHIFT is stuck-at-1, the data will shift into the USER register as normal, but there will be an extra shift during CAPTURE. During CAPTURE, the parallel data input will OR with the shifted data, resulting in data corruption. An indicator of SHIFT stuck-at-1 is during Phase 3A. If the upper 4 bits of the readback results are 0xF instead of 0x1, 0x2, 0x4 or 0x8 (for USER1-4 respectively), it will indicate a SHIFT stuck-at-1.

#### **5.4.4 SEL**

Each Boundary Scan USER Module has a SEL signal that indicates whether that module is active or not. When the tester loads the USER1 Boundary Scan instruction, SEL(1) should be 1 while SEL(4 down to 2) should be 0. The same is true for all other Boundary Scan USER modes. Each step of the test procedure returns a 12-bit number where the upper 4 bits are SEL(4:1).

When in USER1, SEL should be '0001' or 0x1.

When in USER2, SEL should be '0010' or 0x2.

When in USER3, SEL should be '0100' or 0x4.

When in USER4, SEL should be '1000' or 0x8.

If any SEL bit stays unchanged across various USER modes, it indicates a stuck-at fault.

#### **5.4.5 CAPTURE**

If CAPTURE is stuck-at-1, the parallel input PDATAIN will OR with the shifted data on each DRCK. Since one bit in SEL(4:1) is always 1, it will cause at least one bit of the USER register to corrupt.

If CAPTURE is stuck-at-0, both CAPTURE and SHIFT will be low during Boundary Scan Capture. This will cause the new input of all USER registers to be 0. Subsequently shifting data out of the USER register will result in all-zeros returned from TDO.

#### **5.4.6 UPDATE**

Once 12 bits of data are shifted into the USER register, they are latched into the 12-bit Update Latch when Boundary Scan UPDATE signal goes high (during Boundary Scan DR Update). If UPDATE is stuck-at-0, the data will not latch, leaving the 12-bit Update Latch's contents as all zeros. Hence, the lower 8 bits of readback results from Phase 1, 2 and 4 will be all zeros.

If UPDATE is stuck-at-1, the latch will concurrently hold the same data as the USER register. This will result in erroneous results during Phase 3's USER1-3 steps. In Phase 2, the bit 11 shifted into the USER register was 1 to set the PDATAControl bit flag. This flag tells the test circuitry to capture future USER register data from the UAR rather than the Update Latch. During the Boundary Scan Capture stage of Phase 3A/B, however, the USER Register bit 11 will be overwritten by SEL(4), which is 0 for USER1-3. If UPDATE is stuck-at-1, the value 0 in bit 11 will immediately latch into the Update Latch, changing the behavior of PDATAControl from UAR mode (1) to Update Latch mode (0). Hence, the data readback from TDO during Phase 3A and 3B will be Update Latch contents rather than UAR contents.

### **5.4.7 RESET**

The Boundary Scan RESET signal clears the contents of the 12-bit Update Latch to all zeros. Therefore, performing a Boundary Scan Reset followed by a Boundary Scan DR Capture should capture the lower 8 bits of the Update Latch (i.e. all zeros) to the Shift Register.

In Phase 3 of the test procedure, 0xFFF is written to all USER registers. Ordinarily we would expect the USER register readbacks in Phase 4 to return 0xFF in the lower 8 bits, but Phase 4 performs a Boundary Scan Reset before each Boundary Scan DR Capture. Therefore, if the Boundary Scan RESET signal is operational, the lower 8 bits of readback results from TDO will be 0x00 (proving successful clearing of Update Latch contents). If the Boundary Scan RESET signal has a stuck-at-0 fault, the Update Latch will not clear out, and the TDO will return 0xFF in the lower 8 bits.

If the Boundary Scan RESET signal is stuck-at-1, it will keep the Update Latch cleared at all times. Hence, TDO readback's lower 8 bits will always return 0x00 in Phase 1, 2 and 4 (when the USER register captures Update Latch contents) while the TDO readback will return the correct results for Phase 3A and 3B (when the USER register captures UAR contents).

### **5.5 Fault Detection in the User Access Register**

The User Access Register (UAR) is written using configuration memory commands at the beginning of Phase 3A and 3B. During Phase 3A and 3B, UAR contents are captured into the FPGA and read back through the four USER register. Each USER register's lower 8 bits hold 8 of the 32 bits of the UAR.

Prior to Phase 3A of the test procedure, the tester writes the data 0x87654321 to the UAR using configuration memory commands. During Phase 3A, the data is captured into the USER registers 8 bits at a time, and read back through TDO. Next, the tester writes the negation of 0x87654321 (i.e. 0x789abcde) to the UAR using configuration memory commands. During Phase 3B, the data is again captured into the USER registers 8 bits at a time, and read back through TDO. If the original UAR contents are correctly read back in both Phase 3A and Phase 3B, it shows that no stuck-at-0 or stuck-at-1 faults exist in the UAR.

The UAR data\_valid signal goes high for one clock cycle after valid data has been written to the UAR. If the UAR Capture Register is never clocked by data\_valid, it indicates that the data\_valid signal has a fault.

## **5.6 Test Results**

Table 5.1 shows the expected and actual readback results at each stage. The test was successful for LX60, SX35 and FX12, and has been implemented in the Virtex-4 GUI. Table 5.2 shows the size of the compressed configuration bitstream for different Virtex-4 devices. The bitstreams remain fairly small for even the large devices such as LX60.

Table 5.1: Experimental results as expected for LX60, SX35 and FX12

Phase	USR Module	Write (Shift MSB first)	Expected Readback	Actual Readback	Comments
Reset					
1	1	0x011	0x100	0x100	Correct
1	2	0x022	0x200	0x200	Correct
1	3	0x033	0x400	0x400	Correct
1	4	0x044	0x800	0x800	Correct
2	1	0xFEE	0x111	0x111	Correct
2	2	0xFDD	0x222	0x222	Correct
2	3	0xFCC	0x433	0x433	Correct
2	4	0xFBB	0x844	0x844	Correct
Write user access register: 0x87654321					
3A	1	0xFEE	0x121	0x121	Correct
3A	2	0xFDD	0x243	0x243	Correct
3A	3	0xFCC	0x465	0x465	Correct
3A	4	0xFBB	0x887	0x887	Correct
Write user access register: 0x789abcde (negation of 0x87654321)					
3B	1	0xFFF	0x1de	0x1de	Correct
3B	2	0xFFF	0x2bc	0x2bc	Correct
3B	3	0xFFF	0x49a	0x49a	Correct
3B	4	0xFFF	0x878	0x878	Correct
Reset					
4	1	0x011	0x100	0x100	Correct
Reset					
4	2	0x022	0x200	0x200	Correct
Reset					
4	3	0x033	0x400	0x400	Correct
Reset					
4	4	0x044	0x800	0x800	Correct

Table 5.2: Compressed Bitstream Size

Sub-Device	Compressed Bitstream Size (Bytes)
LX60	2,653,523
SX35	2,113,909
FX12	1,058,514

## CHAPTER 6

### SUMMARY AND CONCLUSIONS

This thesis presented a method of injecting faults into the internal resources of an FPGA for the experimental verification of a BIST's fault coverage and diagnostic resolution. The fault injection method partially reconfigured the FPGA to emulate an actual fault. The method used the existing Boundary Scan standard without modifications, and did not present area overhead or design constraints on downloaded FPGA designs. The fault injection method was non-intrusively integrated into the BIST process to test a BIST's accuracy and fault coverage based on a large fault list. This chapter highlights the main contributions of this thesis with respect to prior work, and discusses areas of future research and development.

#### **6.1 Summary and Main Contributions**

This fault injection method was designed and implemented for the Xilinx Virtex-4 series of FPGAs. The method emulated real faults through the partial reconfiguration of the FPGA via Boundary Scan. Since Virtex-4 does not support the partial reconfiguration of a single bit in the configuration memory, the fault injection was performed through a set of configuration memory frame readback, modification and reconfiguration processes (Frame RMW). The Frame RMW procedure was performed

after each BIST configuration download so that the behavior of the FPGA during BIST was consistent with a faulty FPGA.

Even though Frame RMW takes more clock cycles than the direct bitstream fault injection method for ORCA 2C and Xilinx 4000 series previously presented in [4], the Frame RMW has several advantages pertinent to the newer Virtex-4 series. The Slaughter approach was hard-coded to map a small set of FPGA fault sites to bitstream bits. Thus, the Slaughter approach would not work for partial reconfiguration, bitstream compression, or other bitstream options that can change the bitstream structure. On the other hand, Frame RMW works independent of the options used to create BIST configurations since it performs fault injection after the BIST download. Thus, the Frame RMW approach presented in this thesis will work with minimal (if any) modification if Xilinx introduces a new compression technique or other changes to the bitstream structure. Also, it can take advantage of bitstream compression and other measures of reducing BIST download time.

The Frame RMW approach is also an improvement over other fault injection methods such as [7], which only injects faults into the external pins of the FPGA – not the internal resources like Frame RMW. Moreover, Frame RMW does not require FPGA design constraints since it does not use user-defined registers.

The fault injection method described in this thesis hence makes important contributions toward fault injection in the internal resources of FPGAs. It has been verified using the existing Logic BIST, but the method's ability to modify any bit in the configuration memory makes it useful for verifying future BIST for FPGAs as well. This

approach will work with minor modifications on Virtex-5 since Virtex-5 and Virtex-4 have the same basic frame structure.

Table 6.1 summarizes the various Virtex-4 BIST approaches that have successfully used this fault injection emulation technique and includes the number of configurations and number of faults emulated for each BIST approach.

Table 6.1: Number of faults emulated for various BIST approaches

<b>BIST Approach</b>	<b># BIST Configurations</b>	<b># Faults Emulated</b>
Logic BIST	28	976
LUT-RAM BIST	3	52
Block RAM BIST	25	436
DSP BIST	7	156
<b>Total</b>	<b>63</b>	<b>1,640</b>

## 6.2 Future Research

While the fault injection method described in this thesis can dramatically reduce BIST verification time, it has room for improvement. This section discusses potential areas of research and development for the future.

### 6.2.1 Improving Test Analysis when using Partial Reconfiguration BIST

One area in need of improvement is fault injection with partial reconfiguration BIST. As FPGAs increase in size, the BIST configuration download and readback procedures dominate the test time [11]. To reduce the configuration memory download time, newer BIST approaches take advantage of partial reconfiguration. Using partial reconfiguration, the BIST controller only downloads the configuration memory frames

that are different from the last BIST configuration. According to [11], the BIST test time for Xilinx Virtex and Spartan II FPGAs using partial reconfiguration was speeded up 2.86 to 3.61 times, depending on the device version. For larger devices like Virtex-4, the potential gains are even higher.

Partial reconfiguration can dramatically reduce download time, but it has a disadvantage: It does not reset the ORAs between two consecutive BIST configurations. Therefore, once a fault is detected, the ORAs return failure indication even for remaining BIST configurations that did not detect the fault. Even though the presence of ORA failure indications will indicate that some fault was detected, the invalid data in the ORAs will not be useful for diagnosing whether the correct fault was detected. Moreover, it will not be clear which BIST configurations detected the fault since the ORAs will return results with failure indication even for some BIST configurations that did not detect the fault.

One resolution for this issue is to clear the ORAs with each partial reconfiguration. However, this may not be a worthwhile pursuit if the number of clock cycles required to clear the ORAs is close to the number of clock cycles required for a compressed configuration BIST.

### **6.2.2 Fault Injection using Embedded Processor**

The fault injection implementation described in this thesis was a C program running on an external processor. The external processor accessed the configuration memory using Boundary Scan. For BIST where an intelligent BIST controller fully resides on the FPGA as an embedded processor, a fault injection using the embedded

processor would be an important research direction since it will forgo the need for the sending configuration memory commands over the slow serial Boundary Scan connection.

The most significant difference from the external processor implementation would be the use of the Internal Configuration Access Port (ICAP) to access the configuration memory rather than Boundary Scan. ICAP is a 32-bit interface that allows the FPGA fabric to access the configuration memory. Once an ICAP connection is established, the 32-bit commands and data sent to the configuration memory during fault injection should mostly be the same as those currently sent over Boundary Scan. The fault list may still be provided to the embedded processor via Boundary Scan, but the clock cycles required for passing a fault list to the FPGA would be a small fraction of the clock cycles required for Frame RMW of each fault.

## BIBLIOGRAPHY

- [1] Stroud, C E. et al, "BIST of FPGA Interconnect." International Test Conference, pp. 404-411, October 1998.
- [2] Dhingra, S., Milton, D. and Stroud, C. E., "BIST for Logic and Memory Resources in Virtex-4 FPGAs," Proc. IEEE North Atlantic Test Workshop, May 2006.
- [3] Stroud, C.E. "A Designer's Guide to Built-In Self-Test," Boston: Springer, 2002
- [4] Slaughter, T., Stroud, C.E., and Skaggs, B., "Fault Injection Emulator for Field Programmable Gate Arrays", Proc. International Society of Optical Engineering, Vol. 4525, pp. 1-9, 2001
- [5] Xilinx, "Virtex-4 Configuration Guide," UG071 (v1.5), Xilinx, Inc., 2007. Internet: [www.xilinx.com/bvdocs/userguides/ug071.pdf](http://www.xilinx.com/bvdocs/userguides/ug071.pdf)
- [6] Abramovici, M.; Stroud, C.E., "BIST-Based Test and Diagnosis of FPGA Logic Blocks" IEEE Trans. on VLSI Systems, Vol. 9, No. 1, 2001.
- [7] Chakraborty, T.J.; Chen-Huan Chiang; "A novel fault injection method for system verification based on FPGA Boundary Scan architecture" International Test Conference, 2002. Proceedings. 7-10 Oct. 2002 Page(s):923 – 929
- [8] Xilinx, "Virtex-4 User Guide," UG070 (v1.5), Xilinx, Inc., 2006. Internet: [www.xilinx.com/bvdocs/userguides/ug070.pdf](http://www.xilinx.com/bvdocs/userguides/ug070.pdf)
- [9] Vemula, S., and Stroud, C.E., "Built-In Self-Test for Programmable I/O Buffers in FPGAs and SoCs," Proc. IEEE Southeastern Symposium on System Theory, pp. 534-538, 2006
- [10] Xilinx, "Virtex-4 Configuration Guide," UG071 (v1.1), Xilinx, Inc., 2004.
- [11] Dhingra, S., Garimella, S., Newalker, A., and Stroud, C.E., "Built-In Self-Test of Virtex and Spartan II FPGAs using Partial Reconfiguration," Proc. IEEE North Atlantic Test Workshop, 2005.

## APPENDICES

APPENDIX A

LIST OF ACRONYMS

<b>Acronym</b>	<b>Description</b>
ATE	Automated Test Equipment
AUBIST	Auburn University Built-In Self-Test Lab
BIST	Built-In Self-Test
CMD	Command Register (configuration memory)
DR	Data Register
DRCK	Data Register Clock
FAR	Frame Address Register (configuration memory)
FDRI	Frame Data Register In (configuration memory)
FDRO	Frame Data Register Out (configuration memory)
FF	Flip Flop
FPGA	Field Programmable Gate Array
Frame RMW	Frame Read-Modify-Write
GUI	Graphical User Interface
ICAP	Internal Configuration Access Port
IR	Instruction Register
JTAG	Joint Test Action Group
LSB	Least Significant Bit
LUT	Look Up Table

<b>Acronym</b>	<b>Description</b>
MSB	Most Significant Bit
ORA	Output Response Analyzer
PC-III	Xilinx Parallel Cable III
PLB	Programmable Logic Block, a.k.a. Configurable Logic Block (CLB)
RAM	Random Access Memory
RTI	Return to Idle (Boundary Scan state)
STAT	Status Register (configuration memory)
TAP	Test Access Port
TCK	Test Clock (Boundary Scan signal)
TDI	Test Data In (Boundary Scan signal)
TDO	Test Data Out (Boundary Scan signal)
TMS	Test Mode Select (Boundary Scan signal)
TPG	Test Pattern Generator
UAR	User Access Register

## APPENDIX B

### VIRTEX-4 DEVICE IDS

To prevent accidental configuration of the wrong FPGA, configuration memory write operations must supply the correct device ID. Here is a list of Virtex-4 sub-devices, and their respective Device ID codes, as provided in Xilinx's Virtex-4 Configuration Guide [5][10]:

<b>Device</b>	<b>IDCODE</b>	<b>Device</b>	<b>IDCODE</b>	<b>Device</b>	<b>IDCODE</b>
XC4VLX15	01658093			XC4VFX12	01E58093
XC4VLX25	0167C093	XC4VSX25	02068093	XC4VFX20	01E64093
XC4VLX40	016A4093	XC4VSX35	02088093	XC4VFX40	01E8C093
XC4VLX60	016B4093	XC4VSX55	020B0093	XC4VFX60	01EB4093
XC4VLX80	016D8093				
XC4VLX100	01700093			XC4VFX100	01EE4093
XC4VLX160	01718093			XC4VFX140	01F14093
XC4VLX200	01734093				

## APPENDIX C

### VIRTEX-4 BOUNDARY SCAN INSTRUCTION CODES

In addition to performing basic startup and test procedures, Virtex-4 Boundary Scan instructions can be used to access the configuration memory, user-defined registers, the IDCODE register, and other special functions. Here is a summary of Virtex-4 Boundary Scan instructions pertinent to this thesis [5].

<b>Boundary Scan Command</b>	<b>Binary Code (9:0)</b>	<b>Description</b>
USER1	1111000010	Access user-defined register 1
USER2	1111000011	Access user-defined register 2
USER3	1111100010	Access user-defined register 3
USER4	1111100011	Access user-defined register 4
CFG_OUT	1111000100	Access configuration memory interface (output)
CFG_IN	1111000101	Access configuration memory interface (input)
USERCODE	1111001000	Shifts out user code
IDCODE	1111001001	Shifts out IDCODE
JSHUTDOWN	1111001101	Clocks the shutdown sequence