TURTLE: A Fault Injection Platform for SRAM-Based FPGAs

Corbin Alma Thurlow
Brigham Young University

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

Part of the Engineering Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/9025

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.
TURTLE: A Fault Injection Platform

for SRAM-Based FPGAs

Corbin Alma Thurlow

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

Master of Science

Michael J. Wirthlin, Chair
Jeffery B. Goeders
Philip B. Lundrigan

Department of Electrical and Computer Engineering
Brigham Young University

Copyright © 2021 Corbin Alma Thurlow
All Rights Reserved
ABSTRACT

TURTLE: A Fault Injection Platform for SRAM-Based FPGAs

Corbin Alma Thurlow
Department of Electrical and Computer Engineering, BYU
Master of Science

SRAM-Based FPGAs provide valuable computation resources and reconfigurability; however, FPGA designs can fail during operation due to ionizing radiation. As an SRAM-based device, these FPGAs store operation-critical information in configuration RAM, or CRAM. Radiation tests can be performed to prove the effectiveness of SEU mitigation techniques by comparing the SEU sensitivity of an FPGA design with and without the mitigation techniques applied. However, radiation testing is expensive and time-consuming.

Another method for SEU sensitivity testing is through fault injection. This work describes a low-cost fault injection platform for evaluating the SEU sensitivity of an SRAM-based FPGA design by emulating faults in the device CRAM through partial reconfiguration. This fault injection platform, called the TURTLE, is designed to gather statistically significant amounts of fault injection data to test and validate SEU mitigation techniques for SRAM-based FPGAs.

Across multiple fault injection campaigns, the TURTLE platform was used to inject more than 600 million faults to test SEU mitigation techniques, estimate design SEU sensitivity, and validate radiation test data through fault injection.

Keywords: FPGA, Fault injection, SEU mitigation, SEU sensitivity
ACKNOWLEDGMENTS

I would like to thank all those who have made this thesis possible by giving me their support in countless ways. This work would not have been possible without the help and support of all of my invested professors, organizations, colleagues, family, and friends. Their contributions have been invaluable, and the accomplishments of this thesis would not have been possible without them and their support.

My advisor, Dr. Michael Wirthlin, has been vital in guiding me through my research efforts and providing me with the necessary opportunities and tools to advance this work. I am honored that he has given me the opportunity to participate in this work. His encouragement and enthusiasm have inspired my research efforts. I am grateful to the other members of my graduate committee, Dr. Jeffery Goeders and Dr. Philip Lundrigan, and thank them for their support and the knowledge and experience they have helped me gain. I am also thankful for the advice and help of Dr. James Archibald and the other faculty and staff members who have helped me along the way.

My family has greatly supported me throughout my journey. My parents and siblings have also been great supports. The other students in the BYU CCL have also helped me progress in my education and research. Specifically, I would like to thank Andrew Keller, Hayden Rowberry, Andres Perez, Matthew Cannon, Andrew Wilson, Jared Anderson, and Jennings Leavitt. They have always been of great help and aided this work in many ways. I am grateful for their help and for being able to work alongside them.

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 1738550 through the NSF Center for High-Performance Reconfigurable Computing (CHREC). This work was also supported by the Los Alamos Neutron Science Center (LANSCE) which provided neutron beam time under proposal NS-2019-8294-A.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title Page</td>
<td>..................................................</td>
</tr>
<tr>
<td>Abstract</td>
<td>..................................................</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>..................................................</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>..................................................</td>
</tr>
<tr>
<td>List of Tables</td>
<td>..................................................</td>
</tr>
<tr>
<td>List of Figures</td>
<td>..................................................</td>
</tr>
<tr>
<td><strong>Chapter 1</strong></td>
<td><strong>Introduction</strong></td>
</tr>
<tr>
<td>1.1</td>
<td>Thesis Contributions</td>
</tr>
<tr>
<td>1.2</td>
<td>Thesis Outline</td>
</tr>
<tr>
<td><strong>Chapter 2</strong></td>
<td><strong>FPGA Configuration RAM Single-Event Upsets</strong></td>
</tr>
<tr>
<td>2.1</td>
<td>Radiation effects on SRAM-based FPGAs</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Single Event Effects</td>
</tr>
<tr>
<td>2.2</td>
<td>FPGA Architecture and SEU-Induced Failure Modes</td>
</tr>
<tr>
<td>2.3</td>
<td>Measuring the Reliability of FPGA Designs</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Reliability Metrics</td>
</tr>
<tr>
<td><strong>Chapter 3</strong></td>
<td><strong>FPGA SEU Fault Injection</strong></td>
</tr>
<tr>
<td>3.1</td>
<td>General Approach for Fault Injection SEU Sensitivity testing</td>
</tr>
<tr>
<td>3.1.1</td>
<td>General Fault Injection Test Flow</td>
</tr>
<tr>
<td>3.2</td>
<td>General Setup for Fault Injection SEU Testing</td>
</tr>
<tr>
<td>3.3</td>
<td>SEU Sensitivity Testing Fault Injection Platforms</td>
</tr>
<tr>
<td>3.3.1</td>
<td>FT-UNSHADES</td>
</tr>
<tr>
<td>3.3.2</td>
<td>FLIPPER</td>
</tr>
<tr>
<td>3.3.3</td>
<td>SLAAC-1V</td>
</tr>
<tr>
<td>3.3.4</td>
<td>XRTC-V5FI</td>
</tr>
<tr>
<td>3.4</td>
<td>Confidence Metrics for SEU Sensitivity Estimations</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Confidence Intervals</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Standard Deviation of SEU sensitivity</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Coefficient of Variation</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Statistical Analysis of Fault Injection Data</td>
</tr>
<tr>
<td><strong>Chapter 4</strong></td>
<td><strong>TURTLE Fault Injection Architecture</strong></td>
</tr>
<tr>
<td>4.1</td>
<td>General System Architecture</td>
</tr>
<tr>
<td>4.2</td>
<td>TURTLE Architecture Overview</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Nexys Video Board</td>
</tr>
</tbody>
</table>
LIST OF TABLES

3.1 Fault Injection Data for MD5 Encryption Core ................................. 38
5.1 Status Bits .................................................................................. 62
7.1 Base FPGA Utilization of each Benchmark Circuit .............................. 74
7.2 Unmitigated Fault Injection Results ............................................... 75
7.3 B13 Exhaustive Fault Injection Results ............................................ 77
7.4 Fault Injection Geomean Results for PCMF Validation Campaign ........ 80
7.5 Summary of results for single-bit and MCU fault injection ................. 83
7.6 Results for validation of MCU radiation beam testing ....................... 84
LIST OF FIGURES

2.1 Funneling phenomenon in an N-type MOSFET. Adapted from a figure in [1]...........7
2.2 FPGA Architecture layout with various resources. Image adapted from figure in [2]...10
2.3 CRAM defines the LUT contents which defines logic behavior as shown in the gate level image. ................................................................. 11
2.4 SEUs change the CRAM contents which change the implemented logic functions ... 11
2.5 Failures in routing due to SEUs. SEUs can alter correct routing (a) by shorting to ground (b), disconnecting a net (c), or bridging two nets (d) ....................... 12

3.1 Basic Fault Injection Flow .......................................................... 20
3.2 General Setup for Fault Injection SEU Sensitivity testing ............................ 25
3.3 FT-UNSHADES test setup. Figure taken from [3] ................................. 28
3.4 The FLIPPER platform consists of two boards. Images taken from [4] .......... 30
3.5 SLAAC-1V platform diagram. Image taken from [5]. X1 and X2 are the golden design instance and DUT respectively. X0 contains the comparator logic. ................. 32
3.6 XRTC-V5FI test platform. Image taken from [6]. The XRTC motherboard (a), a Xilinx Virtex 5 (b), and a PROM memory card (c) are shown. ....................... 34

4.1 A basic overview of the golden/DUT model ...................................... 41
4.2 The TURTLE architecture includes two FPGAs, one golden copy of the design, and the DUT, which is tested through fault injection ............................ 42
4.3 Two Digilent Nexys Video boards are paired in the TURTLE architecture [7] .... 43
4.4 The FMC coupler card couples the I/O of two Nexys Video boards ............... 44
4.5 FMC I/O configuration through the FMC connector on both the golden and DUT boards .... 45
4.6 The JTAG chain allows the JCM to access and configure both FPGA devices on two separate boards ................................................................. 47
4.7 The JTAG Configuration Manager controls the fault injection process on the TURTLE platform ........................................................................... 48
4.8 A single TURTLE configuration ........................................................... 49
4.9 The multi-JTAG adapter card allows the JCM to control multiple TURTLE configurations ................................................................. 50
4.10 Multiple TURTLEs in a parallel configuration being controlled by a single JCM .... 51

5.1 Different submodules are inserted into each devices for design preparation before testing 56
5.2 Standard BSCAN register cell [8] ......................................................... 60
5.3 Connected BSCAN cells around internal device logic [8] .......................... 61

6.1 Fault injection flow in the TURTLE platform ...................................... 65

7.1 MCU shape reconstruction from radiation data. (a) Observed upset locations from radiation test data. (b) Possible grouping of MCU shapes. (c) Updated MCU shapes based on analysis from [9]................................................. 81
CHAPTER 1.  INTRODUCTION

SRAM-Based FPGAs are often used in terrestrial or space-based applications [10]. Modern FPGAs provide large amounts of programmable logic, computation resources, I/O, memory, and special-purpose functionality. As FPGAs are released on new technology, they can provide this functionality at lower power and operating frequencies. FPGAs are also increasingly integrating dedicated, hardened IP to provide a mix of fixed-function and programmable function solutions. Their reconfigurability and low overhead for design development make them an attractive alternative to application-specific integrated circuits (ASIC).

Static configuration memory (CRAM) cells allow FPGAs to be programmed and used for a variety of purposes. However, CRAM cells are susceptible to upsets induced by ionizing radiation. Values stored in SRAM memory cells can be inverted by a single ionizing particle that passes through the device. This is known as a single event upset (SEU) [11]–[13]. The likelihood of an SEU causing a failure within an FPGA design is commonly referred to as SEU sensitivity. SEU sensitivity is defined as how prone to failure a given design is when a random SEU occurs. It is not guaranteed that an SEU will cause a design operating on an SRAM-based FPGA to fail [14].

Extensive research efforts have been conducted to develop SEU mitigation techniques that aim at mitigating or detecting the effects of SEUs in FPGAs [15]–[18]. These techniques range from triple modular redundancy (TMR), TMR with repair, duplication with compare (DWC), system-modular redundancy, and Error Detection and Correction (EDAC) codes. These techniques attempt to reduce the SEU sensitivity of an FPGA design.

To prove the effectiveness of SEU mitigation techniques, it is common to test these techniques through radiation testing. Radiation testing has two main benefits: it exposes the FPGA to an actual radiation source, causing SEUs to occur at an accelerated rate, and provides different types of radiation testing, such as proton, neutron, and heavy ions [2]. Exposure to actual radiation particles is one of the most important benefits of radiation testing. Testing with actual radiation
particles provides data about the possible radiation effects on the FPGA design and how robust the device is at handling or mitigating those effects. Another significant benefit of radiation testing is that it provides an accelerated environment to collect statistically significant data on the effects of SEUs for SRAM-based FPGA designs. This data can help estimate how the device will behave when deployed for operation in a radiation environment.

However, radiation testing is an expensive and time-consuming process. There are limited locations where radiation testing can be performed, and it can often be a difficult process to obtain time at these facilities for testing. Additionally, large and extensive amounts of preparation need to take place for radiation beam tests. To fully take advantage of radiation beam testing, it is essential to prepare every detail of the radiation test. The time the device is in the beam is critical and costly. Having to troubleshoot issues such as unforeseen failure modes or bugs in test apparatus or setup can be detrimental to a radiation beam test and a potential waste of money.

An alternative option to radiation beam testing to determine SEU sensitivity and test SEU mitigation techniques is through fault injection. Fault injection is the process of artificially introducing an error in the FPGA CRAM cells and then observing the design’s response. Since fault injection introduces artificial errors in the CRAM, it does not replace radiation beam testing. However, it can characterize SEU mitigation techniques and provide SEU sensitivity estimates of the FPGA designs with applied mitigation techniques. Fault injection can also help prepare designs and SEU mitigation techniques for radiation beam tests.

Fault injection offers the ability to quickly introduce faults into the target design for testing the reliability of SEU mitigation techniques. Additionally, fault injection testing does not require as much setup as radiation testing, and custom software or hardware applications can provide a more customized approach for reliability testing. Although fault injection offers a quick and effective method of inserting faults into a design compared to radiation testing, large amounts of fault injection data are needed to provide confidence in calculations of the effectiveness of tested SEU mitigation techniques.

This thesis presents a fault injection platform, named TURTLE, to address both time constraints in injecting faults and the need to collect large amounts of data during fault injection testing. The fault injection platform and methodology presented in this thesis provide a novel approach to perform fault injection across multiple devices to deploy fault injection campaigns that can more
easily collect and analyze large amounts of fault injection data. The platform created in this work uses low-cost off-the-shelf and custom boards to deploy and manage fault injection tests in parallel to collect large amounts of fault injection data for various SRAM FPGA designs.

This fault injection platform also provides a framework to integrate custom FPGA designs for fault injection testing. Additionally, this framework provides the ability to test different configurations of design I/O and clocking techniques to evaluate the effect these configurations may have on the reliability of SRAM-based FPGA designs.

1.1 Thesis Contributions

This thesis presents the TURTLE (Testing Ultra-Reliability Techniques using Low-cost Equipment) fault injection system, a low-cost platform that can artificially induce upsets within SRAM FPGAs. The work presented in this thesis develops a low-cost fault injection system to aid in testing and evaluating the reliability of SRAM-based FPGA designs. The TURTLE platform artificially inserts upsets into the CRAM of both mitigated and unmitigated SRAM FPGA designs to test and evaluate their reliability. This platform was built using both low-cost off-the-shelf and custom-developed boards. The TURTLE platform targeted the Xilinx Artix-7 FPGA by using the Digilent produced Nexys Video board. In addition to testing and evaluating multiple designs, various SEU mitigation techniques were applied to each design and evaluated using the TURTLE fault injection platform.

The TURTLE platform was used to test variations of SRAM-based FPGA designs by implementing non-triplicated I/O, split I/O, and fully triplicated I/O along with other design variations. The designs tested included a B13 finite state machine, MD5 algorithm, AES encryption core, and a SHA3 hash algorithm. Fault injection tests implementing random and targeted fault injection were used to perform tests on the previously mentioned designs. Additionally, various SEU mitigation techniques were applied to these designs and tested through fault injection. The TURTLE verified several SEU mitigation techniques and provided SEU sensitivity estimates of the tested SRAM-based FPGA designs.

This thesis will also discuss the advantages and disadvantages of other fault injection platforms and compare them to the platform presented in this thesis. A general overview of fault injection testing on SRAM FPGAs is presented, and essential aspects of fault injection tools are
covered. The importance of fault injection and a comparison of several fault injection platforms are discussed as well.

From the results obtained through fault injection testing, the TURTLE shows three main contributions. First, the TURTLE platform provided the ability to inject large amounts of faults and provided statistically significant data, which was used to calculate reliability metrics for designs and SEU mitigation techniques. Second, the TURTLE framework facilitated the implementation of several custom FPGA designs and SEU mitigation techniques. These designs and techniques were tested and validated rapidly through fault injection on the TURTLE platform. Lastly, the TURTLE software provided the ease of implementing highly customizable fault injection campaigns, targeting essential bits, specific CRAM frame addresses, and other types of fault injection techniques.

1.2 Thesis Outline

This thesis will first present background information on SRAM-based FPGA architecture and then discuss the effects of radiation on SRAM-based FPGAs in Chapter 2. This chapter will also discuss important reliability metrics used when estimating the SEU sensitivity of SRAM-based FPGAs. Chapter 3 introduces a basic fault injection methodology, a general test setup for fault injection platforms, and a review of other fault injection platforms that have been developed. Additionally, Chapter 3 contains further discussion on specific metrics that are used to convey the quality and confidence of SEU sensitivity estimates achieved through fault injection. Chapter 4 details the TURTLE fault injection platform. This chapter provides an overview of the TURTLE architecture and discusses each component. This chapter also shows how each component works together to perform fault injection tests on the TURTLE platform. Chapter 5 provides additional details on how FPGA designs are prepared and integrated into the TURTLE platform. This chapter will further discuss specific design considerations, constraints, and implemented HDL submodules that aid in integrating custom FPGA designs into the TURTLE platform. The fault injection methodology implemented in the TURTLE platform is described in Chapter 6. Chapter 7 will present multiple fault injection campaigns that were performed using the TURTLE platform. This chapter will also discuss the FPGA designs and SEU mitigation techniques tested in the campaign,
along with the goal of each campaign. Additionally, this chapter will discuss the results from each presented fault injection campaign. This thesis concludes in Chapter 8.
CHAPTER 2. FPGA CONFIGURATION RAM SINGLE-EVENT UPSETS

SRAM-Based FPGAs are susceptible to radiation-induced failures. Most radiation-induced failures in SRAM-based FPGAs result from non-permanent, soft errors that can be prevented, masked, or repaired. Non-permanent errors occur mainly in the device configuration memory (CRAM). These non-permanent errors may alter the FPGA design behavior or cause the FPGA design to cease functioning properly. In preparing designs for use on commercial SRAM-based FPGAs in harsh radiation environments, it is important to understand the sources of radiation, how it affects FPGA designs, and what testing can be done to quantify the reliability of FPGA designs.

Fortunately, many SEU mitigation techniques have been developed to mitigate the effects of radiation-induced SEUs on SRAM-based FPGAs. These techniques make SRAM-based FPGA designs more reliable in radiation environments [10]. In developing SEU mitigation techniques and preparing SRAM-based FPGA designs for use in commercial FPGAs in harsh radiation environments, it is important to understand where radiation comes from, how it affects FPGA designs, and what can be done to improve the robustness of a design to mitigate the effects of SEUs.

This chapter motivates the need for evaluating the effects of CRAM upsets in SRAM-based FPGA designs. Configuration upsets are caused by ionizing radiation, which is present in space and terrestrial environments. These ionizing particles can come from galactic cosmic rays or impurities in packing materials used to develop boards. This radiation can cause FPGA designs to fail or deviate from the desired behavior by upsetting memory values stored in the device CRAM. In SRAM-based FPGAs, the CRAM stored values determine the functionality of the programmable resources within the FPGA. Upsets that occur in CRAM caused by ionizing radiation can cause SRAM-based FPGA designs to fail or deviate from the correct operation. Single Event Effect (SEE) mitigation techniques can be applied to FPGA designs to make them less sensitive to CRAM upsets in harsh radiation environments. This chapter will discuss in further detail single event
effects and their effects on FPGAs, FPGA architecture, and estimating single event upset sensitivity for FPGA designs.

2.1 Radiation effects on SRAM-based FPGAs

Single energetic particles or accumulation of charge from a radiation source can affect SRAM-based FPGAs [1]. When a single energetic particle causes an observable effect within an SRAM-based FPGA, the effect is classified as a SEE. This section provides an overview of radiation effects on SRAM-based FPGAs, focusing on the effects of SEEs and SEUs on CRAM contents of SRAM-based FPGAs. This thesis is focused on analyzing the effects of SEUs on SRAM-based FPGAs, but other SEEs are discussed to provide adequate background information. An explanation of SEEs is given as well as their effect on SRAM-based FPGAs.

Radiation effects occur primarily through the funneling phenomenon shown in Figure 2.1. This figure shows an ionized particle passing through a sensitive node in a semiconductor device. The path of electron-hole pairs is produced from the semiconductor material atoms, resulting in funnel-shaped equipotential surfaces that create a current in the device [1], [19]. This phenomenon can also occur when secondary particles from a neutron collision pass through the sensitive region.
of the device. This phenomenon contributes to radiation effects and is one of the primary sources for SEEs in SRAM-based FPGAs.

2.1.1 Single Event Effects

SEEs are any observable effects caused by single energetic particle [11], [12]. There are two main types of SEEs: destructive and non-destructive. Destructive SEEs often referred to as hard errors, can permanently damage the FPGA. Non-destructive SEEs, often referred to as soft errors, are transient errors. These errors can be removed or repaired in the system or device through means of reconfiguration or complete power cycling of the device. SEEs classified as destructive include: single event latch-up (SEL), single event burnout (SEB), and single-event gate rupture (SEGR). SEEs classified as non-destructive include: single event transient (SET), single event upset (SEU), and single event functional interrupts (SEFI). Of these SEEs, the non-destructive SEEs, or soft errors, are the primary concern for the work presented in this thesis. Of these discussed SEEs, SETs and SEUs are of primary concern for FPGA reliability because they occur more frequently [2], [19], [20].

SELs occur when an internal short between power and ground occurs within the FPGA due to a particle strike. This event can cause permanent damage to the FPGA if the SEL is not removed in time from the device, as the latchup can induce a high current in the device [20]. The fabrication and layout process of the FPGA is a possible method to mitigate this effect [21]. FPGAs can be tested for SEL immunity before being deployed in a harsh radiation environments. Devices that are not immune to SELs are often avoided for use in harsh radiation environments such as space [2].

SETs occur when radiation particles strike particular MOSFET transistor regions, such as the channel region of an inactive n-type MOSFET or the drain region of an inactive p-type MOSFET [19], [20]. A SET induces a current pulse in a MOSFET and appears as a temporary glitch. A SET is assigned a polarity based on the transition of the output of the CMOS device. A positive SET transitions from a logical value of 0 to 1 and back to 0. A negative SET transitions from 1 to 0 and back to 1. SETs can be latched in device memory elements as they propagate through transistors, which may be a larger issue for faster clock speeds in an FPGA design.

SEUs occur when an ionizing radiation particle inverts the stored value in a memory element, such as a bit in the configuration RAM (CRAM) of an FPGA. For example, a single particle
may strike an SRAM cell in an SRAM-based FPGA. A SET could occur within a transistor of the SRAM cell used to maintain cell value. With enough amplitude, this SET can disrupt the SRAM cell and cause it to store the inverted value from the original value in the SRAM cell. The effects of SEUs, as they relate to FPGAs, are further discussed in section 2.2.

Within the CRAM of an FPGA, specific bits are designated control bits for device resources [22]. For example, if an ionizing particle strikes the power circuitry of an FPGA, causing a power glitch which in turn causes a chip reset, this type of event would be a SEFI. A SEFI occurs when a single particle strike affects one of these control elements, causing the entire FPGA to malfunction [20]. While SEFIs are considered soft errors, in some cases, they can prevent the FPGA from being able to reprogram until a complete power cycle is performed. In addition to the control elements external to the FPGA, there are control bits within the FPGA CRAM such as LUTs, routing logic, and memory blocks (BRAMs), that if upset, could disrupt the global functionality of the FPGA. While SEFIs are a concern when considering FPGA reliability, more SRAM-based FPGA design failures are due to SEUs and SETs.

2.2 FPGA Architecture and SEU-Induced Failure Modes

To better understand how SEEs, specifically SEUs, can adversely affect an FPGA design, a more detailed discussion on FPGA architecture is required. This section will describe the basic FPGA architecture and how SRAM-based FPGAs are designed for programmability and design flexibility. In addition to basic FPGA architecture, common SEU failure modes will be discussed. Further examination of SEU failure modes will motivate the need to evaluate FPGA design SEU sensitivity by explaining how an SEU can cause a design to fail.

SEUs are of particular concern for SRAM-based FPGAs because of their large configuration memory. Some newer FPGA devices contain nearly 1 Gb of CRAM [23]. As a comparison, the target Xilinx Artix-7 part used in this thesis work contains approximately 60 Mb of CRAM. The more memory there is within the device, the more frequently SEUs will occur in the device CRAM.

An FPGA is composed of both basic programmable logic building blocks and specialized hard IP blocks. These blocks are organized into a specific architecture within the FPGA. Lookup-tables (LUT) and flip-flops (FF) are grouped into slices or cells. In addition to these resources,
Figure 2.2: FPGA Architecture layout with various resources. Image adapted from figure in [2].

supporting hardware such as carry-chains, multiplexers, and other control signals are included in these cells. Some LUTs are read-only, while others can have their values changed. The ability to change the values in these LUTs gives the FPGA design the flexibility to implement various logic functions depending on the design (e.g., LUTRAMs, shift-registers). Additional hard IP blocks can include the following: large user memories (BRAMs), dedicated arithmetic units (DSPs), analog-to-digital converters (ADCs), high-speed serial I/O (MGTs), and clocking resources (e.g., DCM, MMCM, and PLL). The available resources and architecture are vendor and device-specific, but a common FPGA architecture is an island-style approach. This common architecture indicates that groups of logic resources are surrounded by programmable routing resources and interconnects in a 2D matrix-like formation [24]. This basic formation is shown in Figure 2.2.

Within an FPGA, there exist two major types of bits that store device programming data. The first is configuration RAM (CRAM) bits, and the second is internal block RAM (BRAM) bits. Device programming data that determines the behavior of programmable interconnects, LUTs, control signals, and the contents of smaller user memories (e.g., FFs, LUTRAMs, shift-registers) are stored in CRAM bits. Configuration data associated with larger blocks of user memory are stored in BRAM bits. In addition to these two major types of bits, other types exist within an
FPGA’s programming data, which include global status register and test control logic bits. The configuration and functionality of FPGA designs are loaded into these bits within the FPGA as the design programming data. An example of how the CRAM content defines logic behavior is shown in Figure 2.3.

Since CRAM and BRAM bits store the FPGA circuit configuration and state, SEUs that occur in these bits can alter the behavior or operation of the FPGA design and lead to a potential failure. SEUs that occur in FPGA routing resources, LUTs, control signals, and user memories are the primary cause of SEU failure modes in FPGA designs. SEUs in LUTs and control signals can corrupt user-implemented logic and alter the functionality of primitive blocks used within the FPGA design as depicted in the altered CRAM contents and logic level gates in Figure 2.4. SEUs in user memories can corrupt the state of a design. SEUs in routing resources may cause nets to
disconnect, short to power or ground, or bridge with other nets in the design, causing a change in design operation (see Figure 2.5) [22], [25].

To configure and program the resources and routing interconnects, CRAM cells are used within the FPGA to store this configuration data. Additionally, a large portion of CRAM bits are dedicated to functionality beyond user FFs, BRAM data, and LUT contents [26]. The paper in [25] found through fault injection testing that approximately 80% of FPGA design failures were due to SEUs occurring in the routing resources.

![Figure 2.5: Failures in routing due to SEUs. SEUs can alter correct routing (a) by shorting to ground (b), disconnecting a net (c), or bridging two nets (d).](image)

2.3 Measuring the Reliability of FPGA Designs

A reliable FPGA design can withstand environmental factors, such as radiation, without deviating from the desired or implemented functionality. Unreliability can be a result of faults, errors, or failures occurring within an FPGA design. This section will further discuss these three primary causes of design unreliability. An SEU can cause a fault in the lowest level of an FPGA by corrupting bit-level data used to program the functionality of the device. Within an FPGA
design, an error is the manifestation of a fault within the system. For example, some errors may be detected and will necessitate a recovery, or the error may be ignored if it causes no issue. An error can occur without being considered a failure. A failure occurs when the design deviates from the desired operation [27]. A fault that occurs in an FPGA design may lead to a failure, but the presence of a fault in the system does not guarantee that an FPGA design will fail. A reliable FPGA design can withstand more faults present in the design without failing.

An error is a deviation from the expected design behavior. Some errors can be detected and recovered from or simply ignored. For example, some systems may tolerate a specific number of errors in a derived calculation or some other system operation or design output. These could be classified as acceptable errors. An example of this could be when the checksum of a communication interface fails on the first transmission. However, this triggers a second transmission where the checksum calculation is correct, indicating correct data has been received. Any errors that are classified as unacceptable are considered failures.

A failure occurs when an error causes the design operation to deviate from the desired functionality or defined specification. A failure can lead to corrupt output data that is detectable, as well as silent data corruption. Often failures require global reset operations, complete device reconfiguration, or a full power cycle of the device.

Applying SEU mitigation techniques can reduce the effects of SEUs on FPGA designs. Many post-manufacturing SEU mitigation techniques have been developed for SRAM-based FPGAs. Generally, concepts of redundancy, repair, error detection and correction (EDAC), or the elimination of single-point failures (SPF) are used to design SEU mitigation techniques.

Testing is required to analyze the benefit and improvement offered by an SEU mitigation technique by testing FPGA designs with the applied mitigation technique. There are different testing techniques used to measure the reliability of devices used in harsh radiation environments. These testing techniques can estimate design reliability and validate that an FPGA design will function properly in a radiation environment. Radiation testing is the most common technique used to test mitigation techniques and FPGA designs. Another technique, the main focus of this work, is fault injection. This work focuses on the use of fault injection to measure the effectiveness of various mitigation techniques. The rest of this section will detail various reliability metrics used to measure and analyze the reliability of mitigation techniques and FPGA designs.
2.3.1 Reliability Metrics

Reliability metrics convey the reliability of the tested design or technique and can be used to show how an applied mitigation technique affects the reliability of an FPGA design. These metrics provide insight into how a design may behave in a harsh radiation environment. When performing fault injection on FPGA designs, these metrics help convey the improvement offered by applied mitigation techniques. This section will introduce reliability metrics pertinent to this thesis work and analyze what information is provided through each metric.

Cross-Section

The cross-section is a common metric used to report results from radiation beam tests, which is failures divided by total fluence,

\[ \sigma = \frac{\text{failures}}{\text{fluence}}, \]  

(2.1)

and has units of cm\(^2\). This metric is an average of how many particles per cm\(^2\) it takes to cause a failure. This metric provides the expected number of failures for the system for a specific number of particles. While this metric is commonly used for radiation tests, it is related to both design sensitivity and MTTF, which are discussed further in this section. An example of a cross-section can be shown by comparing an unmitigated FPGA design to the same design but with TMR applied as an SEU mitigation technique. For example, an unmitigated design subjected to radiation testing had a cross-section of \(1.73 \times 10^{-9}\) while the TMR version of the same design had a cross-section of \(1.78 \times 10^{-11}\) [28].

Design Sensitivity

The likelihood of a design failing when a random upset occurs is the SEU design sensitivity of an FPGA design. Other factors contribute to the design sensitivity of an FPGA design, such as the size of the design in terms of device resource utilization and whether or not any SEU mitigation techniques are applied to it.
One of the most important purposes of fault injection is to estimate FPGA design sensitivity to single-event upsets. The sensitivity of an FPGA design is the percentage of CRAM bits, that if upset, will cause the FPGA design to operate incorrectly. This metric can be used to estimate failure rates and the mean-time to failure (MTTF) of a design for different radiation environments.

Design sensitivity can also be estimated through fault injection by injecting a predetermined number of CRAM faults \( n \) into the design and observing the number of system failures \( k \). After injecting a fault, the fault injection system must observe the FPGA design and determine whether the given fault caused any design failures. If the injected CRAM fault caused the FPGA design to deviate from its expected behavior, the CRAM bit is labeled as sensitive. If the injected fault did not cause any problems, it is labeled as insensitive. In most cases, it is not possible to test every CRAM bit, and an estimate of the sensitivity must be made [29]. This sensitivity estimator, \( r \), of the Binomial distribution, can be computed by dividing \( k \) by \( n \) as follows:

\[
r = \frac{k}{n}.
\]  

(2.2)

**Reliability Function \( r(t) \)**

The reliability of a system is the probability that the system is correctly functioning at a given time \( t \). The function output conveys only a probability and does not guarantee that a system is functioning correctly at time \( t \). Any two systems can be compared at any time \( t \), and the reliability function will provide information as to which system has a higher probability of functioning at that time. This metric can be used to estimate the MTTF of a system discussed in the next section.

**Mean Time to Failure**

The MTTF metric represents the average amount of time it takes for the system to fail. The MTTF is the integral of the reliability function previously discussed,

\[
\text{MTTF} = \int_{t=0}^{\infty} r(t) dt.
\]  

(2.3)

This metric provides a single number that can be easily used to compare against other systems. One of the downfalls of MTTF is that it does not convey as much information as the reliability
function. If reliability is more important early in the mission, it would be acceptable to permit a lower MTTF in exchange for higher reliability earlier in the mission. Systems with applied SEU mitigation techniques may have a lower MTTF but provide higher reliability for a shorter time, while unmitigated designs might have higher MTTF but lower overall reliability. Thus, by simply comparing the MTTF metric of two systems, it can be difficult to determine which system would be better suited for a particular system.

These reliability metrics provide the needed measurements to effectively compare the reliability of the designs and applied SEU mitigation techniques tested in this thesis work. Each metric previously discussed has advantages and drawbacks. Collecting and analyzing fault injection using these reliability metrics will provide the foundation through which SEU mitigation techniques can be validated and proven to increase FPGA design reliability.
CHAPTER 3. FPGA SEU FAULT INJECTION

SEU sensitivity estimation is vital in validating SEU mitigation techniques and improving the fault-tolerance of FPGA designs. Mission-critical space applications have rigid requirements for system reliability, and strict requirements must be met to qualify an integrated circuit for space [30]. Part of these qualification tests can include SEU sensitivity testing through radiation and fault injection tests. The benefits of applied SEU mitigation techniques can be proven and verified through SEU sensitivity estimation. Additionally, SEU sensitivity testing may expose other failure modes. The remainder of this chapter will discuss how fault injection can effectively perform SEU sensitivity testing.

To effectively estimate the SEU sensitivity of an FPGA design through fault injection, there must be a method through which failure events caused by an injection, and the related factors causing the failure, can be observed and recorded [2]. Many approaches have been developed to estimate the SEU sensitivity of FPGA designs through fault injection, and a discussion on these related methods is contained later in this chapter. It is important to note that fault injection attempts to mimic the behavior of radiation-induced SEUs on the FPGA design under test by introducing artificial faults into the device CRAM. This chapter will discuss the general approach for SEU sensitivity testing through fault injection on SRAM FPGAs and discuss common components of fault injection platforms. Related work and other fault injection approaches are then presented, along with a discussion about additional metrics used for determining the quality and confidence of reliability metrics achieved through fault injection. The limitations and benefits of the related work are also discussed.

3.1 General Approach for Fault Injection SEU Sensitivity testing

SEU sensitivity testing through fault injection is event-based, where the event is usually the occurrence of a functional failure within the design. Fault injection approaches depend on the
ability to detect when the design under test is not operating correctly [2]. Fault injection testing approaches for SEU sensitivity testing create an environment where a design can fail, and then the platform can measure and record the factors that lead up to failure for analysis. Common elements are found in most fault injection platforms to support SEU sensitivity testing.

Although the mechanism for inserting faults into the design is different for each approach, fault injection artificially introduces faults into the FPGA design. Fault injection can provide essential information on the SEU sensitivity of FPGA designs and provide useful estimations on the effectiveness of SEU mitigation techniques [31].

There are five common elements found in most fault injection SEU sensitivity testing approaches. First is the design under test (DUT). Second, the output of the DUT is usually monitored for errors. Third, the fault injection platform is created to introduce possible failure conditions within the DUT. Fourth, factors leading up to failure, such as the number of injected faults, are recorded for analysis. Finally, a method is implemented to place the design back into a known good state after a failure has occurred to allow for additional failure events to be observed.

In fault injection SEU sensitivity testing, the DUT remains active so that functional failures can occur and be detected. Testing the DUT in all modes of operation is vital during fault injection SEU sensitivity testing to detect failures caused by injected faults. Designs are typically stressed or stimulated through a set of input data vectors [2].

Two common methods exist to monitor the outputs of the DUT for errors. First, the DUT output is compared against a set of generated golden output vectors that match the expected DUT output. Second, the output of the DUT is compared against that of an identical design instance that operates in lockstep with the DUT. This thesis will refer to such a design instance as a golden design instance. If any differences in the outputs of the DUT and golden design instance are found, a functional failure is detected and reported.

Typically, only the DUT is exposed to faults through fault injection. Other components such as a test control unit, functional error detection logic, and the golden output vectors or lockstep golden design instances are commonly separated from the DUT to isolate them from injected faults. For example, fault injection platforms may place the DUT on a separate FPGA. This FPGA is then injected with faults to ensure that no faults are introduced into other platform components, and they only occur within the DUT.
The factors that lead up to a functional failure are recorded for analysis. Additionally, the total number of faults injected and failures observed should also be collected. These data can be used to estimate the SEU sensitivity of the design. Also, the particular logical location of the bit injected before a functional failure may be recorded for reproducibility and failure characterization.

Many failure events must be observed for the collected data to be statistically significant and representative of the SEU sensitivity for a particular design [2], [31]. To properly detect and observe multiple failure events in a single fault injection test, most fault injection testing approaches provide a method of resetting the design. This method will reset the DUT, and any other needed logic, into a known good state so that additional failure events can be properly detected and observed. The DUT is provided with any necessary control or reset signals. The process of resynchronizing or resetting the design for the next test run is often automated to assist in rapid data collection. Most fault injection testing approaches organizes these five common elements into a test setup with an accompanying test fixture to support SEU sensitivity testing. The general fault injection test flow is briefly described in the next subsection.

### 3.1.1 General Fault Injection Test Flow

This subsection will briefly describe the common test flow implemented in fault injection testing. This general test flow, shown in Figure 3.1, consists of seven general steps. These steps are FPGA configuration, fault location selection, fault injection, error detection, device recovery or resynchronization, failure classification, and finally, repairing the injected fault within the DUT.

#### FPGA Configuration

As previously discussed, fault injection targets the DUT, and the first required step in fault injection platforms is to configure the DUT into a known good state. It is essential that after initial configuration, the DUT is actively operating and in a state where no faults are present in the system. It is common to perform an initial status check after configuration on the DUT before injecting any faults. This initial system check ensures the DUT is operating as expected, and any subsequently detected failures from fault injection will be properly classified.
Fault Location Selection

The next step is to select the location in the CRAM where the fault will be injected. Generally, there are two approaches for selecting the fault location. These two approaches are random or targeted selection. Each approach is discussed further in the following subsections.

• Random fault selection

Random fault injection consists of selecting a random frame, word, and bit in the CRAM as the fault to inject (see appendix A for an example). This form of bit selection is similar to what might occur in a radiation test facility and is often used to predict what will happen
during radiation testing. Random fault injection has an equal likelihood of upsetting any bit within the CRAM of the FPGA.

- **Targeted fault selection**

  Targeted fault injection selects specific CRAM bits or device tiles to inject as specified by a user-generated file or some other user implemented method [32], [33]. This form of bit selection is often used to “replay” upsets collected at a radiation test facility. An example of targeted fault injection could include only injecting faults into user flip-flops or injecting faults into essential or critical bits.

  Another method of targeted fault injection is exhaustive or sequential fault injection. An exhaustive fault injection test injects a fault into every user-accessible CRAM bit, testing each bit for sensitivity. Similar to an exhaustive selection, sequential fault injection injects faults sequentially from the first specified location to the last specified location. This form of bit selection can be used to exhaustively test all CRAM bits or specific addresses within the CRAM.

  Both approaches offer advantages and disadvantages. Random fault injection is often used to mimic the random nature of SEUs observed during radiation beam tests. Random fault selection is also easy to implement as the random location is selected from a range of valid CRAM addresses within the DUT. However, random fault injection may not be the best selection method. Targeted fault injection can provide a more controlled approach to fault injection by selecting specific CRAM bits to inject and provide the ability to replay SEUs collected from radiation tests.

**Inject Fault**

After the fault location selection, the fault is injected into the DUT. This injection is performed by inserting the fault into the DUT CRAM. It is common for the fault injection platform to implement some method to confirm the fault has been injected successfully.

For example, after inserting the fault into the DUT CRAM, it is common to read the same CRAM address immediately following the injection. Reading the same CRAM address and confirming the bit value has changed ensures the fault was inserted successfully.
Additionally, the fault injection step may require an added delay after injecting the fault. This delay may be used to allow the fault to propagate through the system before performing error detection. Allowing the fault to propagate in the system for more time may be necessary to allow errors to propagate to the output of the design.

**Error Detection**

As previously discussed, there are two main approaches to monitor DUT output signals and detect errors. The first approach compares the DUT output to a golden set of predetermined output data that matches the expected output of the DUT. For fault injection, this output vector could be stored internally in the device or externally and compared off-chip to determine if a has occurred in the output.

The second approach compares the DUT output against the golden design instance output running in lockstep with the DUT. The output from both designs is compared, detecting any differences in the output and reporting errors as necessary. While these are two common approaches, there exist other methods for detecting errors. For example, platforms may monitor design-produced checksums, internal state registers or may implement other external methods of output data comparison.

**Device Recovery**

To accurately detect and observe all failure events during fault injection, the fault injection flow needs to provide a method to recover or reset the DUT into a known good state of operation. This often means performing a resynchronization to reset the DUT and ensure the DUT is free of faults.

This device recovery could also be performed through a complete DUT reconfiguration or power cycling of the device if necessary. This behavior is essential to ensure that no failure event caused by a previous injection is logged or counted in subsequent injections. This ensures that additional failure events can be properly observed and logged during a single fault injection test.
Classify Failure

When a failure event is detected, the event must be logged and classified. It is common for failures to be classified into different categories based on the severity of the failure caused by the injected fault. For example, some failures may be fixed by simply restoring the injected bit to the original value, or the failure may require a complete DUT reconfiguration to eliminate it from the system and restore the DUT to a known good state.

These failures need to be classified and logged during fault injection to understand how the DUT responded to the injected faults. This data can also be used to determine what SEU mitigation techniques may be helpful to eliminate such failures. This information is critical to accurate SEU sensitivity testing during fault injection. The data that is logged and classified during this step could include such information as the injected bit location or address within the DUT CRAM and the severity classification that the failure caused.

Additionally, during the fault injection test, it is essential to record the total number of injected faults and the total number of detected failures. The logging of these bits allows for future tests to better reproduce failures in the DUT and characterize failure modes. This data collection and classification ensures that the SEU sensitivity estimate calculated from the fault injection results is accurate.

Most fault injection platforms follow a similar test flow as described in this subsection. It is common for each platform to customize or adapt each step in the fault injection flow for fault injection tests based on varying needs or requirements. The general test setup for fault injection platforms is described in the next section before discussing related work.

Repair Fault

Repairing the injected fault means restoring the injected CRAM bit to the original value written to the SRAM cell during the initial configuration. The injected fault may be repaired at different stages of the fault injection flow as shown in Figure 3.1. This repair may occur after an error has been detected and classified, or when no error is detected, it should be repaired before proceeding with further injections. The important aspect of repairing the fault is to restore the
DUT into a known good operating state. Repairing the injected fault and ensuring no failures are present in the DUT before injecting more faults is essential to the fault injection flow.

3.2 General Setup for Fault Injection SEU Testing

In addition to the similar test flow discussed in the previous section, the test setups for many fault injection platforms share common aspects. Figure 3.2 depicts a general setup for an SEU fault injection platform and common test components, which are as follows:

- Host Computer
- External Interface
- Error Detection Mechanism
- Design Under Test (DUT)
- Golden Output Vectors or Gold Design Instance
- Clocking and Control Signals
- Test Control Unit

Each test setup typically has a host processor or host computer that interfaces with the test fixture, which generally consists of the FPGA device and other test apparatus. The host processor or computer can control the overall test flow, enable and disable the injection of faults into the DUT, and issue global resets when necessary. Typically, the host computer handles data logging and ensures that every failure event detected by the error detection mechanism is properly logged. It is common to include automated data logging on the host or controller device, watchdog or heartbeat monitors to ensure the DUT is operating, and automated selection algorithms for fault injection location selection on the host computer.

The external interface from the DUT to the host computer or processor provides the ability to monitor the internal status of the DUT during fault injection testing. Examples of this external interface could include Ethernet connection, memory interface, or serial interface. While the exact
For fault injection testing, detecting errors in the DUT is vital to determine when functional errors cause the design to fail. An error detection mechanism provides the ability to detect functional errors in the DUT in real-time. In fault injection testing, this mechanism may implement error detection by comparing generated golden output data or output from a golden design instance to the DUT output. If an error or mismatch of output data is detected on the DUT output, the failure is classified and logged. Accurate error detection is vital in fault injection testing because it directly impacts the data used to calculate accurate SEU sensitivity estimates and other metrics used to characterize and analyze SEU mitigation techniques.

The error detection mechanism compares the DUT output to the generated golden output or golden design instance output. If a golden design instance is running in parallel to the DUT, these designs must execute in lockstep with each other. Lockstep operation ensures that all operating design instances produce the expected outputs in the same order. Since FPGA designs often have
predictable data flow paths, it is possible to achieve lockstep operation for a pair of FPGA designs. In addition, FPGA designs operating in lockstep must begin execution on the same clock cycle to ensure they operate in parallel. For FPGA designs with more complex data-flow paths such as out-of-order computation, these designs may require more complex methods to keep the designs operating synchronously.

The test fixture holds the FPGA DUT, where faults will be introduced into the DUT CRAM. Many of the other test components discussed in this section can be combined into the same FPGA design. However, the DUT refers specifically to the logic that is tested through fault injection. If the test FPGA contains additional logic outside of the DUT logic, the fault injection platform may target specific CRAM addresses during fault injection to specifically target the DUT logic. It is also common to separate the DUT from any additional test components. This approach isolates the DUT from other design components, that if upset, could potentially cause a failure during fault injection not related to the DUT operation. This isolation approach ensures that any fault injected into the test device will affect only the logic implementing the DUT. This approach is commonly implemented by placing the DUT on a separate FPGA.

The generated golden output, or gold design instance output, is compared against the DUT output to determine when failure events occur. Golden output vectors may be stored within the FPGA BRAM, but other approaches may store these generated output vectors off-chip. Other approaches may implement a golden design instance on the same FPGA device as the DUT, and these two designs run in lockstep. Additionally, platforms may implement the golden design instances on completely separate FPGA devices to isolate them and ensure no faults are introduced into the golden designs during fault injection testing.

Clock and control signals drive the DUT and may control when input stimulus is provided to the DUT. These control signals typically provide methods to reset or resynchronize the DUT after an observable failure event. The host computer may trigger these clocking and control signals to the rest of the test fixture, or they may be triggered internally when specific failure modes are detected during testing. Additionally, clocking signals may be generated from external interfaces such as a separate FPGA device with a golden design running in lock-step with the DUT. These shared clock signals ensure lock-step operation.
A test control unit controls the DUT operation, controlling how the DUT interfaces with other components and the rest of the test fixture. It could generate input stimulus or ensure the DUT operates in lockstep with a golden design instance if used. The test control unit also commonly relays important status through the external interface to the host computer. This status could include notifying the host computer when to log specific data or when the DUT is in an unrecoverable state and needs to be reconfigured for device recovery.

These are commonly implemented components of fault injection platforms and provides basic information on how fault injection platforms operate. Each component is usually modified, customized, or tailored for the specific needs of the fault injection platform. The following section will describe various fault injection platforms and their fault injection approach and platform components.

3.3 SEU Sensitivity Testing Fault Injection Platforms

The fault injection platforms presented in this section focus on estimating SEU sensitivity of SRAM-based FPGAs through both radiation and fault injection testing. These platforms focus on estimating the reliability benefits of SEU mitigation techniques. While most of the presented platforms have performed SEU sensitivity testing for FPGAs in radiation testing, this thesis will focus on their fault injection testing approaches. The reliability benefits of SEU mitigation are estimated by testing both mitigated and unmitigated designs and comparing the estimated SEU sensitivity of designs through fault injection.

Fault injection and radiation testing are both common methods for estimating the SEU sensitivity of SRAM-based FPGA designs. SEU sensitivity, previously discussed in Chapter 1, is defined as how prone to failure a given FPGA design is when an SEU occurs. For fault injection testing, the CRAM bits are injected with faults to emulate SEUs in the design. A target design operates on the FPGA while faults are injected into the CRAM bits. One of the primary indicators to determine the SEU sensitivity of a design in a fault injection test is the number of design failures that occur with respect to the number of injected faults.

In addition to previously discussed reliability metrics, an additional metric, known as mean upsets to failure (MUTF), can be used to measure the SEU sensitivity of a design. However, it is also common to measure the SEU sensitivity as a percentage of upsets that cause the design to fail.
When considering the MUTF of an FPGA design, a lower MUTF or higher percentage of sensitive bits means that the FPGA design is more susceptible to SEUs. For fault injection, SEU sensitivity is commonly discussed as design sensitivity, and for radiation testing, it is typically discussed in terms of cross-section.

The rest of this section discusses related fault injection platforms used to estimate SEU sensitivity. This section will describe the test setup and discuss results from experiments performed on each platform. The limitations and benefits of each approach are discussed and compared.

### 3.3.1 FT-UNSHADES

The study in [34], used the platform, named FT-UNSHADES [3], which emulates fault injection in ASIC hardware using an FPGA as a fault injection emulation platform. This approach uses formal verification to ensure that the ASIC design and the corresponding FPGA design are equivalent. This platform uses partial reconfiguration to inject faults into the user flip-flops in the

![Figure 3.3: FT-UNSHADES test setup. Figure taken from [3].](image)
design. This platform implements a golden and DUT instance of the design on the same FPGA. In addition to these designs, comparison and test control logic are implemented on the same FPGA. Input stimulus test vectors are loaded from an external device, and faults are injected into flip-flops. For this platform, injected faults are limited to flip-flops in the DUT. Figure 3.3 shows the test setup implemented in the FT-UNSHADES platform. This fault injection platform uses a single FPGA to implemented the DUT and golden design. Faults are emulated on the single-FPGA platform along with other required logic.

This particular approach tested a LEON2 soft processor through fault injection. This approach tested an unmitigated, XTMR (Xilinx automated TMR) and fault-tolerant (LEON3-FT) design. The study in [3] selected the most sensitive components of the processor to test and analyze during fault injection. The fault injection approaches described in [35] and [36] is similar to that of FT-UNSHADES.

This platform uses a lockstep golden design instance implemented on the same FPGA as the target DUT tested in fault injection. This approach differs from the platform presented in Chapter 4 as the TURTLE platform implements the golden design instance and DUT on separate FPGAs. This platform also includes test control logic and error detection on the same device. Another difference between this platform and the one presented in Chapter 4 is this approach limits injected faults to the specific CRAM addresses used by the DUT. The TURTLE can target any user-accessible CRAM bits during fault injection.

Since this approach limits injected faults to flip-flops, only a small portion of the FPGA architecture is included in the SEU sensitivity estimation from data obtained through fault injection. This approach does not directly consider the effects of SEUs that occur in logic or routing resources in an FPGA design as it only targets flip-flops. However, a major difference between this approach and the one presented in this thesis is that only a single FPGA is needed to implement this platform. Implementing both the DUT and golden design instance in a single FPGA makes it less expensive to implement. This platform achieves fault injection times between 1.4 ms and 1.6 ms, similar to the work presented in this thesis.
3.3.2 FLIPPER

The FLIPPER is a fault injection platform that injects faults using partial reconfiguration [4]. This platform also uses input stimulus and generated golden output data values derived from HDL simulation for use in error detection. These predetermined values are compared to the DUT output in real-time to detect failures in the DUT. If the DUT output does not match the expected golden data values for the given input stimulus, a failure has occurred in the DUT. The FLIPPER platform consists of two separate boards: the mainboard that manages the fault injection algorithm and other aspects of the test, and the DUT board, which contains the FPGA design to operate under the presence of faults in the system.

![Main Board and DUT Board](image)

Figure 3.4: The FLIPPER platform consists of two boards. Images taken from [4].

In a case study using the FLIPPER, faults were injected at a rate of 18 ms per injection. This rate of fault injection includes the time for partial reconfiguration of the CRAM frame, which takes 50 $\mu$ seconds. The additional time in the 18 ms per injection is comprised of the functional test execution time for the DUT to complete 26,000 testbench cycles, generate fault data, and communicate the data to the host. With this approach, the MUTF of various mitigation schemes was evaluated for an FPGA design. Similar to the results presented by the FLIPPER platform, the TURTLE platform evaluated multiple SEU mitigation techniques across different FPGA designs and estimated the SEU sensitivity for those designs and the applied SEU mitigation techniques.
The plain design, without mitigation, had a MUTF of 337 accumulated faults. The Xilinx TMR version had a MUTF of 1,330 accumulated faults. Based on these results, Xilinx TMR without repairing injected faults reduced the SEU sensitivity of the evaluated design $3.9 \times$. 

Similar to the TURTLE platform presented in this thesis, the FLIPPER uses two separate test boards. This approach allows for the ability to extend the fault injection platform to other devices and helps reduce reoccurring engineering costs for future fault injection testing and adapting to other devices [31]. One major difference between the FLIPPER platform and the TURTLE is that while they both use two separate boards for fault injection testing, the FLIPPER board uses two different FPGA devices. The mainboard is a Virtex 2 Pro, and the DUT board is a Virtex 2. The TURTLE platform uses two identical Xilinx parts, which aids in simplifying the implementation of the golden design instance and the DUT on a single FPGA part. This architecture decision is a benefit of the TURTLE as it eliminates additional design time required to implement two designs on two different devices.

3.3.3 SLAAC-1V

The SLAAC-1V platform is an FPGA fault injection platform that uses three FPGAs [5]. The three FPGAs used in the platform each have different functional purposes for SEU sensitivity testing. One FPGA implements the DUT that will be tested in either fault injection or radiation testing. Another FPGA functions as a golden design instance that runs in lockstep with the DUT and is designed to be the FPGA that is fault-immune. Lastly, the other FPGA implements test control logic. This FPGA includes logic for data comparison to detect failures, reconfiguration, and control logic of the other two FPGA devices.

This approach is similar to the TURTLE by using off-chip error detection and using the control FPGA to compare output from complex designs without having to precompute golden output data. The DUT and golden design instances receive the same input stimulus, which should lead to both running design instances to produce the same output, except in the case of a failure present in the DUT. The TURTLE platform is similar to this platform, as a separate FPGA device implements error detection and data comparison where the DUT is not operating. In both this platform and the TURTLE, the golden design instances run on separate FPGA devices to isolate the DUT logic for fault injection testing.
The testing procedure using the SLAAC-1V platform was able to inject a fault every 215 µs. Between each fault injection, both the DUT and golden design instance continue to operate. The output data from both designs are compared on the third device to detect differences between the DUT and golden design instance. This work performed an exhaustive test on all 5.9 million CRAM bits in the Xilinx device.

The SLAAC-1V platform conducted another experiment where collected proton radiation test data was taken and used for fault injection testing to correlate SEU sensitivity estimates between both methods [37]. This experiment demonstrates another important role fault injection plays in SEU sensitivity testing. The fault injection data shows that SEU sensitivity estimates achieved through fault injection testing agreed with radiation test data results. The data shows that through fault injection, a 36-bit and 72-bit multiplier, and a 72-bit LFSR circuit, demonstrated a 4.0%, 14.7%, and 4.8% SEU sensitivity estimation respectively for each design. In radiation testing, these same designs had a 4.9%, 17.2%, and 5.0% sensitivity estimation.

Several SEU mitigation techniques were tested and compared in [16] using the SLAAC-1V board. This approach gathered fault injection data across multiple SEU mitigation techniques. This
experiment simulated CRAM configuration scrubbing, or CRAM upset correction, by correcting the injected fault after each injection. This configuration scrubbing prevents the accumulation of upsets in the CRAM after each fault injection.

One advantage of this platform is that there are two separate FPGAs, one for the golden instance design and one for the DUT, which allows for two complex FPGA designs to run in lockstep. For complex FPGA designs, data comparison between the output data of two FPGA designs operating in lockstep can be easier with two separate FPGA devices and designs than comparing the output of a single FPGA against golden output data. This platform differs from FT-UNSHADES and the TURTLE platform as it uses a third FPGA to implement the comparison logic to detect errors in the DUT output. This approach allows faults to be injected anywhere on the device without the risk of injecting CRAM bits related to logic outside of the DUT implementation, disrupting other test or control logic. This approach also allows the SLAAC-1V platform to test designs in radiation testing. The TURTLE platform uses a single FPGA to implement the golden design instance and the comparison logic.

### 3.3.4 XRTC-V5FI

The work in [6], presents the XRTC-V5FI fault injection platform consists of three main boards: the XRTC motherboard, the Xilinx Virtex 5 FPGA DUT card, and an external memory card shown in Figure 3.6. The motherboard contains two FPGAs: the ConfigMon and the FuncMon. The ConfigMon FPGA controls all of the reconfiguration and fault injection of the DUT card, and the FuncMon implements failure detection of the DUT output. This platform differs from other platforms by comparing the output of two identical design instances on the FPGA DUT card rather than comparing the DUT output to an external golden design instance or generated golden data values to detect failures. The output of two running design instances on the FPGA DUT card are compared against each other to detect failures during fault injection. Output signals are sent to the FuncMon FPGA and then compared to determine if a failure has occurred. In this platform, even though two design instances are implemented on the same FPGA, it is not likely that a single injected fault will affect both design instances in the same way. Additionally, since the comparison logic is separate from the FPGA containing the DUT, it is unlikely that injected faults will cause the error-detection logic to fail.
The XRTC-V5FI board has been used in several fault injection experiments. In [6], a fault injection experiment evaluated the SEU sensitivity of various soft-core processors. For this fault injection test, the test exhaustively injected each bit on the DUT FPGA card for SEU sensitivity testing. The soft-core processors were tested with several different benchmark programs. In [38], this experiment tested other circuits through fault injection and then correlated fault injection data with radiation test data. This experiment found that data from fault injection and radiation data showed similar results in SEU sensitivity estimation. Additionally, this platform injected over 5 billion configuration bits that required thousands of hours of testing. This fault injection platform achieved a fault injection rate of one injection every $49.1 \mu s$.

The XRTC-V5FI is similar to both the FLIPPER and TURTLE fault injection platforms. The DUT is implemented on a separate FPGA for injecting faults. This approach reduces reoccurring engineering costs and allows the DUT to be injected freely without the risk of causing failure in other test component logic. The XRTC-V5FI differs from the previously discussed SLACC-V1
approach by placing identical FPGA design instances on the same FPGA. This approach reduces the number of target FPGAs needed to operate lockstep designs. However, it also reduces the available logic for DUT implementation as both design instances must fit on the same FPGA together.

3.4 Confidence Metrics for SEU Sensitivity Estimations

While general reliability metrics have previously been discussed, two final metrics are presented in this section to show the importance of measuring the quality and confidence of the SEU sensitivity estimate obtained through fault injection. Two metrics used to convey the confidence and quality of SEU sensitivity and reliability metrics are confidence intervals and the coefficient of variation. These metrics provide further insight into the estimated metrics obtained through fault injection.

It is important to remember that fault injection only provides an estimate of SEU sensitivity. Large amounts of fault injection data must be collected to increase the quality and confidence in SEU sensitivity estimates. The two metrics discussed in the following subsections expand further on the need for collecting large amounts of data during fault injection to obtain better confidence in the previously discussed reliability metrics in section 2.3.

3.4.1 Confidence Intervals

All previously discussed reliability metrics are probabilistic in nature, and as such, require standard confidence intervals to convey the quality and confidence of the reliability estimations. Typically 95% confidence intervals are used, which convey a range of confidence in the statistic or measured value. This confidence interval is the 95% likelihood that the actual measured value, such as SEU sensitivity measurement, lies within the range defined by the confidence interval. When calculating confidence intervals, tighter bounds or interval bounds convey greater confidence that the actual value or measured metric is relatively close to the estimated measured value. Confidence intervals are calculated based on methods presented in [2] and depend on the number of faults injected and failure events observed. For fault injection testing, the number of events corresponds to the number of failures observed during a fault injection test. The more events observed, the tighter the bounds on the confidence intervals will be.
When more than 50 events are observed, the standard 2 sigma standard deviation can be
used as follows,

\[ 2\sigma = \frac{2\sqrt{\text{number of events}}}{\text{number of experiments}}. \]  

(3.1)

For fault injection tests, the “number of events” would be the number of failures observed. 
The “number of experiments” would be the total number of injections performed during the fault
injection test. The total number of injections directly corresponds to the number of CRAM bits
tested in fault injection. The 2 sigma error is used to calculate lower and upper confidence bounds
as follows,

\[ \text{bound} = \text{value} \pm 2\sigma. \]  

(3.2)

The equation calculates the upper and lower bound based on the estimated measured value.
For fault injection, it is common to calculate the upper and lower bounds on the estimated design
sensitivity based on the number of failures detected and the number of faults injected. For fault
injection tests where fewer than 50 events, or failures, are observed, then the lower and upper
bounds must be calculated separately. In this case, the upper and lower bounds can be determined
using the tables shown in Figure 22 taken from [2].

3.4.2 Standard Deviation of SEU sensitivity

SEU sensitivity for FPGA designs can be estimated with fault injection by injecting a prede-
termined number of CRAM faults \((n)\) into the design and observing the number of system failures
\((k)\). After injecting a fault, the fault injection system must monitor the FPGA design and determine
whether the injected fault caused any design failures. If the injected CRAM fault caused the design
to deviate from its expected behavior, the CRAM bit is labeled a sensitive bit. If the injected fault
did not cause any problems, it is labeled an insensitive bit. In most cases, it is not possible to
test every CRAM bit, and SEU sensitivity must be estimated. This sensitivity estimator, \(\hat{r}\), of the
Binomial distribution, can be computed by dividing \(k\) by \(n\) as follows:

\[ \hat{r} = \frac{k}{n}. \]  

(3.3)
A large number of faults is needed to obtain an accurate estimate of the sensitivity. The standard deviation of the sensitivity estimator is calculated with the following equation:

\[
\sigma = \sqrt{\frac{k}{n^2} \left( 1 - \frac{k}{n} \right)}
\]  

(3.4)

The standard deviation of the sensitivity estimator reduces by \(1/n^2\), suggesting that a large number of faults must be injected to obtain statistical confidence in the estimate.

### 3.4.3 Coefficient of Variation

A metric related to confidence intervals is the coefficient of variation [39]. This metric shows the extent of the variability in the target metric in relation to the mean population. In fault injection testing, the coefficient of variation can determine the variability in reliability measurements, such as SEU sensitivity or design sensitivity estimations. Both the standard deviation of the sensitivity estimator and the sensitivity estimator discussed in the previous section are needed to calculate the coefficient of variation for design sensitivity estimations.

Once the standard deviation has been calculated as discussed in the previous section, based on the number of faults injected and failures observed, this value can be used to calculate the coefficient of variation of the design sensitivity as follows,

\[
\text{Coefficient of Variation} = \frac{\sigma}{\hat{r}}.
\]  

(3.5)

This metric is used to compare fault injection SEU sensitivity estimates across different SEU mitigation techniques applied to FPGA designs. This metric shows that you need many more fault injections for higher confidence in the SEU sensitivity estimate for a design with TMR applied than non-TMR.

### 3.4.4 Statistical Analysis of Fault Injection Data

One important aspect of estimating these metrics is that sufficient amounts of data are needed to calculate increasingly accurate estimations. Obtain accurate design sensitivity estimations requires a large number of faults to be injected into the DUT. As SEU mitigation techniques
are developed that successfully reduce the SEU sensitivity of FPGA designs, it is more difficult to obtain statistically accurate estimates of the design sensitivity as fewer failure events may be observed during testing. To obtain statistically accurate estimates, FPGA designs with lower sensitivity (i.e., those with SEU mitigation) must have significantly more faults injected than FPGA designs with higher sensitivity (i.e., those without SEU mitigation).

<table>
<thead>
<tr>
<th>Description</th>
<th>Unmitigated</th>
<th>TMR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faults Injected ((n))</td>
<td>2,000,000</td>
<td>14,000,000</td>
</tr>
<tr>
<td>Failures ((k))</td>
<td>129,677</td>
<td>486</td>
</tr>
<tr>
<td>Estimated Sensitivity ((\hat{r}))</td>
<td>6.48%</td>
<td>3.5E-3%</td>
</tr>
<tr>
<td>Coefficient of Variation</td>
<td>2.69E-3</td>
<td>4.54E-2</td>
</tr>
<tr>
<td>95% Confidence Interval</td>
<td>(6.45%, 6.52%)</td>
<td>(3.2E-3%, 3.8E-3%)</td>
</tr>
</tbody>
</table>

The need for large numbers of fault injection can be demonstrated with the SEU mitigation technique of Triple Modular Redundancy (TMR). TMR is a well-known SEU mitigation technique that involves triplicating circuit resources and voting on the results [40]. TMR will mask errors from one copy of the triplicated circuit when the other two circuit copies give a correct result. TMR has been applied to FPGA circuits and has been shown to significantly reduce design sensitivity. As an example, this technique was applied to an MD5 encryption core. Table 3.1 demonstrates the effectiveness of TMR with the results of a fault injection campaign completed on this platform. The unmitigated design has a sensitivity of 6.48\%, and with TMR, the sensitivity reduced to \(3.5 \times 10^{-3}\%\) (over 1000x reduction in design sensitivity).

Although the TMR design has a much lower estimated sensitivity, the coefficient of variation of the estimate is much higher than that of the unmitigated design. This data conveys the need for large amounts of fault injection data to obtain better confidence in design sensitivity estimations. Compared to the unmitigated design, the TMR design had approximately 7x the number of faults injected. However, the TMR coefficient of variation is over 20x larger than that of the unmitigated design, conveying less confidence in the estimated sensitivity. More injections and
observed failures are needed to reduce this coefficient and obtain tighter interval bounds in the estimated design sensitivity.

The example from Table 3.1 demonstrates the need for a fault injection platform capable of providing a method to inject a large number of faults. The TURTLE platform was designed to address this need by providing a low-cost platform that can inject large amounts of faults into FPGA designs. The primary goal was to design a fault injection platform that could collect statistically sufficient data with as little cost as possible.
CHAPTER 4. TURTLE FAULT INJECTION ARCHITECTURE

The TURTLE platform architecture was created with three main goals. First, to provide a low-cost approach for artificially injecting faults within an SRAM FPGA. Second, generate large amounts of fault injection data, and lastly, allow the user to more easily customize fault injection tests. These goals were the driving factors for the architecture design decisions made when creating the TURTLE platform.

Due to the cost of the components used to create the TURTLE, this platform implements fault injection at a relatively low cost. Other platforms incorporate expensive custom built boards, custom developed PCBs, or other expensive test fixture apparatus. The TURTLE is low-cost because it uses off-the-shelf FPGA development boards, with low-cost small, custom PCBs. By combining several components, all of which will be discussed in this chapter, the TURTLE is built using low-cost FPGA development boards and other components. The FPGA development boards and other components allow the TURTLE architecture to utilize multiple SRAM-based FPGAs to implement parallel fault injection tests at a low cost.

To accurately test, validate, and understand how SEU mitigation techniques improve SRAM-based FPGA design reliability, large amounts of fault injection are needed. As previously stated, due to the low-cost nature of the TURTLE platform, multiple fault injection tests can be run simultaneously, which allows the TURTLE to achieve the goal of collecting statistically significant amounts of data.

The TURTLE provides a framework through which FPGA designs can be integrated into the platform for fault injection testing. This framework implements necessary functionality, such as error detection, I/O handling in the FPGA, test control unit, and data generation. Additionally, the software developed on the TURTLE allows for customizable and adaptable fault injection testing approaches. This chapter will provide details on the TURTLE architecture and discuss key design decisions made to achieve the desired goals of the fault injection platform.
4.1 General System Architecture

The TURTLE architecture was built following the golden/DUT model. This model, shown in Figure 4.1, implements two separate FPGA designs. One design, denoted as the golden design, is the FPGA design operating correctly with no errors or failures. The golden FPGA design displays the correct behavior and produces correct outputs during operation. The design under test (DUT) is the identical FPGA design implemented in the golden FPGA design. These two identical designs operate in lockstep, as shown in Figure 4.1. To test the DUT, faults are injected into the DUT CRAM while both designs operate and function from the given input stimulus. The outputs produced from the golden design instance and DUT are compared each clock cycle to determine if they match. As the DUT is injected with faults, it is important that output from both devices are compared each clock cycle to detect any possible error in the DUT output and to ensure no failures are missed.

This golden/DUT model has both advantages and disadvantages when compared to the platforms presented in Chapter 3. One advantage of this approach is that it is relatively easy to implement the circuits and designs for the golden design instance and DUT. This implementation is relatively easy because the golden design instance and DUT are the same design implemented
on the same device. This approach requires little to no extra development to adapt either the golden design instance or DUT to a different device. A disadvantage of this approach is that more FPGA logic is required: one FPGA to implement the golden design and another FPGA to implement the DUT logic. The details of how this golden/DUT model is implemented in the TURTLE platform are given in the following sections.

4.2 TURTLE Architecture Overview

The golden/DUT model is implemented in the TURTLE platform as shown in Figure 4.2. This model consists of a golden FPGA and a test FPGA connected with a custom FPGA Mezzanine Card (FMC). The golden FPGA contains a golden design instance (golden DUT) which will run in lockstep operation with the DUT on the test FPGA. The golden FPGA also implements a method to supply input data to both the golden DUT and DUT on the test FPGA. Additionally, the golden FPGA contains comparison logic for detecting errors on the output data of the DUT from the test FPGA. Finally, faults are injected, and system status is monitored through the JTAG Configuration.

Figure 4.2: The TURTLE architecture includes two FPGAs, one golden copy of the design, and the DUT, which is tested through fault injection.
Manager (JCM) [41]. Additional details on each component will be provided in the following subsections.

### 4.2.1 Nexys Video Board

The TURTLE architecture implements the golden design instance and DUT on separate FPGA devices. This approach requires two FPGA boards for a single TURTLE configuration as the platform uses commercial boards rather than a custom-built board. The TURTLE uses the Digilent Nexys Video board, a low-cost FPGA prototyping board used primarily for video related circuits [7] (see Figure 4.3). The Nexys Video board contains a single Xilinx Artix-7 series FPGA (Xilinx part number XC7A200T-1SBG484C), which contains 215,360 logic cells (32,650 slices). The FPGA implements a variety of I/O interfaces such as Ethernet, USB, HDMI video, and PMOD connectors.

![Image of Nexys Video board](image)

**Figure 4.3:** Two Digilent Nexys Video boards are paired in the TURTLE architecture [7]

An important component of the Nexys Video board used in the TURTLE platform is the FMC connector. This board implements the 160-pin FMC low-pin count (LPC) standard connector and is used in the TURTLE system for transmitting control signals and test data between the golden
design and DUT. The two FPGA boards are coupled through this FMC connector by a custom FMC coupler card described in the next section.

The Artix-7 part on the Nexys Video board can be configured through different methods, such as USB-JTAG, an onboard EEPROM, a microSD card, and a JTAG interface using a 6-pin JTAG through-hole port. The external JCM device connects to the Nexys board using the 6-pin JTAG header for programming, monitoring, and testing purposes.

4.2.2 FPGA Mezzanine Card

A custom-built printed circuit board (PCB), shown in Figure 4.4, was designed to connect two Nexys Video boards using the FMC connector. This FMC coupler card provides a mechanical connection between the FMC connectors of the two Nexys Video boards. Two Nexys Video boards connected with a single FMC coupler card compose a single TURTLE system. This card also provides onboard clocks that are distributed to both FPGA boards. This clock architecture allows both the golden design instance and DUT to run synchronously in parallel during fault injection. This

Figure 4.4: The FMC coupler card couples the I/O of two Nexys Video boards
feature is a key aspect of coupling the two designs together. This synchronous operation allows the golden design to receive output from the DUT and compare the output of the golden design and DUT every clock cycle. This fine-grained data comparison enables better error detection and ensures any failures caused by the injected faults in the DUT CRAM are accurately logged.

The FMC coupler card uses 32 differential I/O pairs for a total of 64 data signals between the two FPGA devices. While the LPC connector provides 160 physical pins, the LPC specification only requires 68 of those to be data lines. The remaining signals are dedicated to JTAG, I2C, power, ground, and other purposes. The data signals between the two boards are configured inversely to the other board. This configuration allows for the same FPGA design to operate on either board without additional I/O customization to properly transmit and receive data.

Figure 4.5: FMC I/O configuration through the FMC connector on both the golden and DUT boards

Figure 4.5 shows the I/O configuration and how the signals are connected inversely to each other, connecting signal 0 on the golden board to signal 63 on the DUT and so on. In addition to the 64 data signals, the FMC coupler card provides three reset signals, two control signals, and up to three synchronous clock signals for both FPGAs.
The three reset signals provide the ability to individually reset a single TMR domain when testing a DUT that has an applied TMR SEU mitigation technique. These reset signals can be applied through software written on the external controller or host computer. They can also be triggered internally by the golden FPGA if necessary to restore the proper functionality of the DUT.

Similarly, the three clock signals can be implemented to supply individual clocks to each domain within a TMR FPGA design. These three clocks are generated on the FMC coupler card and are sent to the FPGA design on three separate clock-capable I/O signals.

The PCB design was designed carefully for the layout and routing of the clocking and reset signals (see appendix C for further details). These signals were implemented carefully to ensure accurate timing and that both the clocking and reset signals interfaced identically to both FPGA devices. The FMC coupler card also links the JTAG chain of both boards. The JTAG signals are routed through the FMC coupler card, which places both FPGA devices on a single chain accessible to the JCM. The JTAG interface is described in the next section.

4.2.3 JTAG Interface

The JTAG interface is crucial to the TURTLE platform. The TURTLE injects faults, monitors system status, and controls the fault injection flow through the JTAG interface. While faster interface methods exist for device configuration or device access, many off-the-shelf boards do not support all of these methods. JTAG was selected for this platform because of its wide use in development boards such as the Nexys Video board. Also, the JTAG standard provides the ability to control multiple through a single JTAG chain.

The ability to chain multiple devices on a single JTAG chain is an important aspect of the TURTLE platform. By placing multiple devices on a single JTAG chain, a single device can configure and access each device through a common connection. The FMC coupler card unifies the JTAG chain of each device to provide a single JTAG chain. Figure 4.6 shows how the JTAG chain connects the two FPGAs to form a single chain. For the TURTLE, a single JCM can access and configure both devices from the single JTAG chain as shown in Figure 4.6. The JTAG chain, beginning with the JCM, first connects to the golden device, then the chain continues through the
The JTAG Configuration Manager, or JCM (see Figure 4.7), controls the fault injection process implemented on the TURTLE platform through custom software and hardware. The JCM is a Linux-based embedded system implemented on the Zynq-7000 programmable SoC and was designed to generate high-speed custom JTAG command sequences [41]. The programmable logic
The PL portion of the Zynq device contains a custom hardware circuit for controlling the JTAG TAP controllers. This dedicated circuit for controlling the JTAG chain allows the system to provide high-speed JTAG communication and the flexibility to create various JTAG command sequences. The JCM connects to a custom PCB carrier card to the board containing the Zynq-7000 programmable SoC. This custom carrier card has one JTAG port connection and a 40-pin expansion I/O connector that the TURTLE platform uses for fault injection testing.

The JTAG interface provides the JCM configuration access to both devices, allowing the JCM to control the fault injection test (see Figure 4.6). Through custom-developed software, the JCM can configure either FPGA to perform initial configuration, inject faults into the DUT CRAM, repair CRAM faults, read configuration status and other device registers, and perform complete reads of device CRAM contents (see Appendix B for example code). The JCM can also send or receive data from the golden FPGA through the BSCAN registers implemented in the design.

The JCM also contains a network interface through TCP/IP protocol. In a typical configuration, an external host computer connects to the JCM through an Ethernet interface allowing the user to access the JCM Linux system through an SSH connection to execute the fault injection code. After executed fault injection campaign is complete, the user can transfer fault injection logs from the JCM to the host computer for post-processing.
Figure 4.8 shows a complete single TURTLE system composed of two Nexys Video boards, coupled by an FMC coupler card, connected to a JCM. As shown in the figure, the TURTLE is composed of relatively inexpensive FPGA boards and commodity components.

4.3 Parallel TURTLE Configuration

As previously discussed in section 3.4.4, large amounts of fault injection data are needed to obtain higher confidence in the SEU sensitivity estimation through fault injection. The TURTLE platform can implement parallel fault injection tests to inject larger amounts of faults over a shorter amount of time. When compared to a single TURTLE platform, parallel fault injection tests provide the obvious advantage of collecting more data in the same time frame and allow for simultaneous testing of different design variations by targeting multiple FPGA devices.

Several fault injection tests can execute and operate simultaneously on several single TURTLE instances placed into a parallel configuration. These tests are deployed and monitored simultaneously by a single JCM. Since a single TURTLE is a relatively low-cost system, expanding the TURTLE platform is also a low-cost system. Single TURTLE instances can be placed in a parallel configuration called a “pond”. The JCM controls and operates the parallel fault injection tests in the TURTLE pond through the multi-JTAG adapter card discussed in the following subsection.
4.3.1 Multi-JTAG Adapter Card

The TURTLE platform was designed to be a scalable platform. This scalability increases the ability to collect statistically significant amounts of fault injection data. An essential component that allows the TURTLE platform to be scalable is the multi-JTAG adapter card. This adapter card, shown in Figure 4.9, attaches to the 40-pin expansion I/O header on the custom PCB carrier card as part of the JCM platform.

The multi-JTAG adapter card allows the JCM to control up to six single TURTLE instances in parallel operation. The individual JTAG ports on the adapter card can operate in parallel, allowing the JCM to control, monitor, and run multiple fault injection tests simultaneously on each available JTAG port. This feature is a key aspect of the TURTLE system as it allows the platform to inject large amounts of faults into multiple devices. The JCM can control multiple fault injection tests, injecting faults into the CRAM of multiple DUTs through the JTAG interface.

![Figure 4.9: The multi-JTAG adapter card allows the JCM to control multiple TURTLE configurations](image)

4.3.2 TURTLE Pond

The JCM and multi-JTAG adapter card allow for up to six fault injection tests to operate simultaneously to increase the number of injections performed and data collected. This final platform enhancement enabled the TURTLE to achieve the goal of collecting statistically significant amounts of data through fault injection testing for more accurate SEU sensitivity estimations. The
statistically significant data collected through the parallelized TURTLE platform enabled better testing and validation of SEU mitigation techniques through fault injection testing. Multiple TURTLE ponds can test various DUTs simultaneously through fault injection. Different single TURTLe instances of the TURTLE ponds can test different DUTs, or the same DUT can be deployed and tested on several layers.

These large amounts of fault injections are vital to collect data needed to calculate important reliability metrics for tested SEU mitigation techniques. Large amounts of data are especially needed to test and validate more robust and improved SEU mitigation techniques. More fault injections need to be performed on these higher reliable SEU mitigation techniques to observe and collect more failure events to obtain higher statistical confidence in design sensitivity estimation and reliability improvement from the SEU mitigation technique.

In addition to performing large amounts of fault injection, the TURTLE platform and hardware framework allows for easy implementation of various designs. These designs can be deployed simultaneously across multiple TURTLE platforms. Furthermore, the hardware configuration and communication implementation between the DUT and golden design instance allows for the relatively easy implementation of these designs for fault injection testing on the TURTLE platform.
Additionally, the software used in the TURTLE provides the ability for highly customizable fault injection campaigns.

The software developed and used for the TURTLE can be customized for various types of fault injection campaigns. For example, designs and SEU mitigation techniques can be easily tested by targeting essential design bits or injecting specific CRAM frame addresses. Additionally, the software can be easily modified to replay data obtained from radiation tests.

The next chapter will discuss the HDL framework created as part of the TURTLE platform to integrate custom FPGA designs into the TURTLE platform for fault injection testing. The next chapter will cover specific design considerations that should be reviewed when integrating an FPGA design into the TURTLE platform. Additionally, the next chapter will provide further implementation details on both the golden and test FPGA.
CHAPTER 5. TURTLE FPGA DESIGN PREPARATION

In the TURTLE platform, special design preparations and considerations must occur before a design is ready to be tested with fault injection. These considerations include meeting design timing and I/O constraints that must be met during design implementation. Additionally, specific submodules need to be inserted, shown in Figure 5.1, into both the golden device and DUT. These design considerations and submodules will be discussed further in the following sections.

5.1 Design Considerations

To ensure that both the golden and test FPGA designs (see Figure 5.1) function correctly, they must meet proper timing and I/O constraints. As previously discussed in section 4.2.2, the FMC coupler card provides clocking signals to each design. Additionally, data is sent between the I/O of the FMC coupler card to both the golden and DUT FPGA designs. When preparing a design for testing on the TURTLE platform, it is essential that design timing is met and that the proper I/O constraints are implemented.

5.1.1 Timing Constraints

Both the golden and test FPGA designs must meet timing constraints to ensure that no errors or failures detected during a fault injection test are from improper design implementation or failure to meet timing constraints. Failure to meet timing restraints could result in improper data transmission from the DUT to the golden FPGA, resulting in mismatching data causing an error or failure to be detected.

In this thesis work, third-party vendor tools, such as Xilinx Vivado, were used to perform timing analysis and identify any design-critical paths that cause timing issues. Meeting timing constraints in both the golden and DUT design is essential to ensure that the golden design instance
runs in lock-step with the DUT design. Any unresolved timing issues in either design could result in false errors being detected during fault injection.

As previously discussed in section 4.2.2, it is possible to send up to three clock signals to each design from the FMC coupler card. The global clock on the FMC coupler card that drives these clock signals runs at 100MHz. The timing constraints and implementation for each design may vary depending on if one or three clocks are implemented. However, either approach requires the designs to meet timing requirements.

An additional challenge to consider when implementing FPGA designs with added SEU mitigation techniques will typically involve handling added logic to the FPGA design from applying the SEU mitigation technique. For example, when applying TMR as an SEU mitigation technique, the DUT will contain additional logic elements to triplicate the desired circuit. This circuit triplication, and thus additional FPGA design logic, may require different approaches or steps to meet timing requirements when implementing such designs.

Since applying SEU mitigation techniques involves adding additional logic to a design, other issues may occur when attempting to place and route a design. For example, when triplicating an FPGA design, routing or placement congestion can occur. This routing or placement congestion may require custom placement constraints in the FPGA design and may necessitate additional design constraints to meet timing requirements.

There are additional approaches to resolving timing issues within an FPGA design. For the TURTLE platform, approaches such as adjusting design placement constraints and adding pipeline stages to data paths aided in resolving timing issues. Additional testing and design simulation helped verify that the golden design instance and DUT operated in lockstep, producing the same output every clock cycle. Another vital part of generating an FPGA design for the TURTLE platform is to implement proper I/O constraints.

5.1.2 I/O Constraints

Section 4.2.2 describes the FMC coupler card used to couple the I/O of the golden and DUT FPGA designs for proper data transmission. Figure 4.5 in section 4.2.2 also shows how usable 64 data signals are connected for data transmission and reception between the two designs. For proper
data transmission on both the golden and test FPGA designs across the FMC coupler card, proper I/O constraints must be implemented during design placement and routing.

As Figure 4.5 in section 4.2.2 depicts, these I/O constraints are constructed to allow the golden and DUT designs to properly function on either FPGA in a single TURTLE configuration without needing to regenerate either design with new I/O constraints. As previously discussed in 4.2.2, there are only 64 pins available for data transmission between the golden and DUT FPGAs. This limitation requires that the designer consider what data needs to be transmitted during fault injection testing.

For example, some designs may have an output of greater than 64 bits. In this scenario, reduction networks could be implemented in either the golden or DUT designs to reduce the amount of data sent across the FMC coupler card to the 64-bit limit. This reduction network could be as simple as an XOR operation of specific data, with this XOR reduced data being the actual data sent across the FMC coupler card.

Additionally, designs implementing TMR could consider partitioning the 64-bit data lanes into three separate data busses, each transmitting identical data to an individual TMR domain in the DUT FPGA. The I/O constraints and configuration can be adapted for fault injection tests with different needs or requirements.

These I/O constraints allow the TURTLE to more effectively implement various designs without the need to recreate the I/O constraints and configurations every time a new FPGA design is created and integrated into the TURTLE platform. These constraints can be easily applied to different designs or easily modified from the base set of constraints when needed.

5.2 Design Submodules

In addition to the design considerations discussed in the previous section, the golden and DUT FPGA designs require different design submodules for proper functionality. Figure 5.1 shows the submodules contained in the golden FPGA design, which are:

- Golden DUT
- Master Finite State Machine (FSM)
- Data Generator
• Error Detection
• JTAG BSCAN

Figure 5.1: Different submodules are inserted into each devices for design preparation before testing

The golden design instance is an exact copy of the DUT tested during fault injection. The golden design instance will run in a lock-step operation with the DUT placed on the test FPGA. The Master FSM plays a vital role in synchronizing the golden design instance and the DUT to ensure data is sent, received, and compared correctly. The data generator is responsible for generating and transmitting data to both DUT instances. The error detection module receives output responses from both the golden design instance and DUT and performs comparator operations to detect failures in the DUT output, and logs test status to the JTAG BSCAN registers. As seen in Figure 5.1, the DUT, or test FPGA, primarily contains the DUT logic that will be tested through fault injection with minimal additional design logic for handling resets and I/O data signals.

These submodules create an HDL framework through which a custom DUT and golden design instance can be placed into the TURTLE platform to perform fault injection testing. This framework eliminates the need to recreate the submodules that handle key functionality in the TURTLE platform as part of the golden FPGA design, as shown in Figure 5.1. Additionally, the
minimal logic implemented on the test FPGA allows custom DUTs to be implemented with little to no modification for proper handling of data or control signals between the golden and test FPGA.

While these submodules provide a baseline framework to integrate custom DUTs into the TURTLE platform, they also allow for modifications as needed or required for any specific fault injection test or DUT. Figure 5.1 shows each submodule, and the following subsections will discuss the role they play in the TURTLE platform.

5.2.1 Design Under Test

In both the golden and test FPGAs, a majority of available design logic is dedicated to implementing the DUT. As seen in Figure 5.1, the golden DUT and test DUT are placed on separate FPGAs. The DUT on the test FPGA is the design that will be targeted and tested through fault injection. Both the golden and test DUT receive identical data during testing.

As discussed in the previous section, the DUT placed on the test FPGA may have additional logic implementing a reduction network or perhaps serializing the transmitted data if the DUT response exceeds the 64-bit data limit across the FMC coupler card. Both DUTs will generate output responses based on the input data, and these output responses will be sent to the error detection submodule. To ensure that both DUTs operate in lockstep, a master FSM controls certain aspects of the fault injection flow discussed in the following subsection.

5.2.2 Master Finite State Machine

An additional submodule implemented in the golden FPGA is an FSM that aids in synchronizing the two DUTs. Once both the golden and test FPGA have been initially programmed, the master FSM aids in synchronizing the operation of both DUTs. Synchronization between the DUTs is vital to ensure proper failure classification and detection during fault injection testing.

Upon initial device configuration or during operation, the master FSM can assert reset signals to the DUT on the test FPGA if needed. For example, the master FSM will assert the reset signals after device initialization to reset the DUT on the test FPGA to ensure both the golden DUT and DUT on the test FPGA begin lock-step operation. Additionally, the master FSM could apply
resets after an injected fault has been repaired, but a failure continues to persist. The master FSM has four states, which are initialize, reset, delay, and compare.

**Initialize**

The initialize state initializes reset and control signals when both devices are programmed for a fault injection test. This state ensures that both the DUTs on the golden and test FPGA are operating in lockstep.

**Reset**

The reset state helps synchronize the designs as well as issue internal resets when necessary during fault injection. Both the golden DUT and DUT on the test FPGA have synchronous reset signals.

**Delay**

The delay state is used in the golden FPGA to determine when the output response from the DUT is valid after device initialization.

**Compare**

The compare state enables the golden FPGA to compare output responses from both devices and determine if a failure has occurred.

Once initialization has taken place, and the master FSM has determined that both the golden design instance and DUT are operating in lockstep, the master FSM will continue to operate in the compare state. The compare state will enable a control signal sent to the error detection unit to convey that data received on each following clock cycle will be valid data to compare.

During normal operation, once the master FSM is in the compare state, the master FSM will only change states if the JCM asserts an external reset or if the master FSM detects that the DUTs are no longer operating in lock-step. The JCM could assert a reset signal for different reasons. For example, if the JCM detects a persistent error even after repairing an injected fault, the JCM may send this reset could be applied. This external reset would transition the master FSM to the reset state and trigger a resynchronization of both the golden DUT and test DUT. The master FSM may also transition to the reset state during operation if it detects that the DUTs are no longer synchronized. The main role of the FSM is to ensure the DUT and golden device operate
synchronously, applying resets when needed after device configuration, initialization, or during normal fault injection testing.

5.2.3 Data Generator

The data generator submodule is responsible for providing input stimulus data to the golden DUT and DUT on the test FPGA. The data generator is controlled by the master FSM, which applies an enable signal to the submodule when data should be generated and sent to the golden design instance and DUT. The data generator may use a Linear-feedback shift register (LSFR) or other data-generating logic to generate random data. This data stimulus is applied to the golden DUT and sent across the FMC coupler card to the DUT on the test FPGA.

As previously discussed in Chapter 3, there are different approaches to providing input stimulus to the DUT. While the data generator may supply pseudo-random data from logic implemented by an LSFR, it is also possible to store predetermined input stimulus into BRAM. Depending on design and test requirements, the data generator submodule may be a simple FSM that continuously reads data from BRAM, given a specific address, and then applies that input to the DUTs. This approach can be easily implemented through the TURTLE framework as well as by generating pseudo-random data.

5.2.4 Error Detection Logic

To achieve synchronous lockstep comparison between the responses of the golden device and DUT, the golden FPGA implements an error detection submodule. For each input test vector, the error detection logic receives corresponding output responses (see Figure 5.1) from both devices. These responses are registered and compared on a cycle-by-cycle basis. The responses are compared to determine if there is an error in the DUT response. An error is detected when the DUT response does not match the one given by the golden DUT.

As previously discussed in section 5.1.2, the I/O across the FMC coupler card may be partitioned into data responses from separate TMR domains when TMR has been applied to the DUT. The error detection submodule receives this I/O and performs comparisons of each domain
response. It will also perform a majority vote of two out of three responses from the three TMR domains to determine if the entire DUT has failed.

This data comparison provides a more fine-grain error detection when implementing designs with applied SEU mitigation techniques. For example, if TMR has been applied to an FPGA design, the error detection submodule can provide more fine-grained logging details. The error detection submodule can log which domain failed and convey if the entire design failed or if only one TMR domain had an error on the output. This test status is conveyed from the error detection submodule to the JTAG BSCAN interface described in the following subsection.

5.2.5 User BSCAN Interface

The TURTLE platform takes advantage of the IEEE JTAG BSCAN standard for data logging and system status. The BSCAN primitive is a design element that can be used in Xilinx FPGAs to provide access to internal logic through JTAG chain access [42]. Figure 5.2 shows the standard JTAG BSCAN shift-register cell for each signal pin of the device that is accessible through the boundary scan interface.

![Standard BSCAN register cell](image)

This standard shift-register cell allows for data to be both read and written through the BSCAN interface to the FPGA design. The golden FPGA implements these BSCAN primitive registers
using custom HDL code to make the system status and internal data signals available through JTAG Chain access.

Figure 5.3 shows how the standard shift-register previously discussed are connected in a dedicated path around the device’s internal logic components. This dedicated register path allows the JCM to query and monitor internal data signals with the FPGA device. In the TURTLE platform, the JCM uses the BSCAN registers to perform both reads and write. Custom HDL within the FPGA design determines how to interpret the write data in the BSCAN registers and applies the data to the FPGA design. The JCM uses the BSCAN registers for two main functions: Asserting reset signals to the golden design instance and DUT and monitoring the DUT status as determined by the error detection submodule during fault injection.

![Figure 5.3: Connected BSCAN cells around internal device logic](image)

The golden FPGA design stores and updates the system status in the BSCAN registers, making the data available to be read by the JCM. The golden FPGA in the TURTLE platform uses a 32-bit system status further described in Table 5.1. The least significant byte in the 32-bit value
is updated when failures are detected during fault injection. The DUT status bit indicates a DUT failure and is set to 1 when a failure is detected. The lower 7 bits of this byte contain information about individual domains when implementing TMR designs. The other three bytes in the system status are a predetermined constant value. This predetermined constant value is determined by the golden FPGA design. When the lower byte is all zeros, and the rest of the system status reflects the predetermined constant, the system is in a known good state with no errors.

Table 5.1: Status Bits

<table>
<thead>
<tr>
<th>Name</th>
<th>Bit Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Constant</td>
<td>[31:8]</td>
<td>Serves a verifiable tag for the JCM to identify the golden FPGA</td>
</tr>
<tr>
<td>DUT Status</td>
<td>7</td>
<td>Indicates whether the DUT output matches the output from the golden DUT</td>
</tr>
<tr>
<td>Constant</td>
<td>6</td>
<td>A constant zero value</td>
</tr>
<tr>
<td>TMR Domain 2</td>
<td>5</td>
<td>Signifies if the output from domain 2 in the DUT matches the golden DUT</td>
</tr>
<tr>
<td>TMR Domain 1</td>
<td>4</td>
<td>Signifies if the output from domain 1 in the DUT matches the golden DUT</td>
</tr>
<tr>
<td>TMR Domain 0</td>
<td>3</td>
<td>Signifies if the output from domain 0 in the DUT matches the golden DUT</td>
</tr>
<tr>
<td>TMR Domain 2</td>
<td>2</td>
<td>Signifies if the output from domain 2 in the DUT matches the majority voted value from all three TMR domains in the DUT.</td>
</tr>
<tr>
<td>TMR Domain 1</td>
<td>1</td>
<td>Signifies if the output from domain 1 in the DUT matches the majority voted value from all three TMR domains in the DUT.</td>
</tr>
<tr>
<td>TMR Domain 0</td>
<td>0</td>
<td>Signifies if the output from domain 0 in the DUT matches the majority voted value from all three TMR domains in the DUT.</td>
</tr>
</tbody>
</table>

The specific design considerations and submodules discussed in this chapter provide the guidance and framework for integrating custom designs into the TURTLE platform. The design
considerations for timing and I/O constraints are essential for proper design functionality in the TURTLE platform. Additionally, these constraints and considerations allow the TURTLE to be more versatile by using these constraints as baseline considerations for every design integrated into the TURTLE platform for fault injection. These considerations can be adopted for each design as needed, allowing the TURTLE to accommodate a variety of both FPGA designs and applied SEU mitigation techniques.

The submodules discussed in this chapter provide an HDL framework that implements test-critical functionality. This HDL framework provides an easier approach to integrating custom FPGA designs into the TURTLE platform by eliminating the need to recreate these functional blocks for every new FPGA design or new SEU mitigation technique applied to an FPGA design. Each submodule has a role in the TURTLE fault injection methodology discussed in the next chapter.
CHAPTER 6. TURTLE FAULT INJECTION METHODOLOGY

The fault injection approach implemented in the TURTLE system uses custom software and hardware on the JCM. This section describes the fault injection algorithm implemented on the TURTLE platform, as shown in Figure 6.1. The flow explained in the section is a fault emulation methodology that injects faults into the DUT CRAM. The TURTLE methodology follows the basic fault injection flow described in section 3.1.1, by implementing the following basic fault injection steps:

• Initialization
• Fault Location Selection and Injection
• Design Execution with input stimulus
• Error Checking and Fault repair
• Data logging and failure classification

These steps form the TURTLE fault injection methodology implemented by the TURTLE framework, the JCM software, and hardware. Each step in the fault injection methodology is discussed further in the following sections.

6.1 Initialization

The first step in a fault injection test on the TURTLE is initialization. An important aspect of initialization is calibrating the operating clock speed of the JTAG chain. Speed calibration is performed to determine the maximum operating clock speed for data transfer through the JTAG chain. Calibration for maximum speed must account for cable length and the number of devices on the JTAG chain. The maximum data transfer speed determined during calibration affects the
overall fault injection speed. After calibration of the JTAG clock speed, the golden device and DUT are programmed with bitstreams through JTAG.
After the devices are programmed, the two designs must be synchronized to ensure both circuits receive and process the same test vectors on the same clock cycle. The master FSM in the golden FPGA helps with the synchronization process previously discussed in section 5.2.2. The master FSM applies internal resets to synchronize the golden design instance and DUT and ensure they are in lockstep operation. This synchronization is essential to run both designs in lockstep and compare output results from the golden design instance and DUT each clock cycle.

Once the system status shows the designs are synchronously operating, the fault injection test can proceed. Upon successful initialization, the methodology will select the fault injection location and enter a loop as indicated by Figure 6.1. This operation loop will continue until the fault injection test is terminated, or there are no more faults to be injected as part of the fault injection test. The steps performed during fault injection in this continuous loop are discussed in the following sections.

### 6.2 Fault Location Selection and Injection

After initialization, software on the JCM selects a CRAM bit where the fault will be injected (see Figure 6.1). As previously discussed in section 3.1.1, the two main methods for selecting fault location are random and targeted. The TURTLE implements multiple methods of fault injection that can be classified under these two main categories. Under the random category, the TURTLE implements a baseline random selection, and under targeted, the TURTLE can implement essential, sequential, and multi-cell fault injection selection.

- **Random Fault Selection**
  
  Random fault injection consists of selecting a random frame, word, and bit in the CRAM as the fault to inject. This form of bit selection is similar to what might occur in a radiation test facility and is often used to predict what will happen during radiation testing.

- **Targeted Fault Selection**
  
  Targeted fault injection selects specific CRAM bits to inject. This form of bit selection may be used to “replay” upsets observed at a radiation test facility by injecting the collected upset data into the DUT CRAM during fault injection testing.
Essential

Essential fault injection can inject and target specific bits in the bitstream that have been designated as essential by a third-party vendor tool such as Xilinx Vivado. This form of bit selection is helpful if the test needs to limit the number of targeted bits or if the FPGA designer wants to understand how the essential bits affect the design sensitivity.

Sequential

Sequential fault injection injects faults sequentially in the bitstream from the first location to the last location. Additionally, this form of bit selection can exhaustively test a full device bitstream by injecting every possible CRAM location.

Multi-Cell

Multi-cell fault injection injects multiple faults into the bitstream according to predetermined multi-cell fault injection patterns. This bit selection is used to more accurately emulate what occurs during radiation where multiple CRAM bits are upset by a single ionizing particle.

After selecting the fault location within the CRAM, the JCM injects the fault through partial reconfiguration. To inject the fault, the JCM performs the following steps:

1. Read the target CRAM frame(s) addresses from the DUT via JTAG Chain access,
2. Invert the value at the target frame, word, and bit within the CRAM data,
3. Write the corrupted CRAM data using partial reconfiguration to the DUT, and
4. Read the target frame, word, and bit to verify the fault injection.

6.3 Design Execution

As part of the fault injection methodology, once the fault is injected and present in the system, the DUT must operate on all given input test vectors generated from the data generator submodule on the golden FPGA before the fault is removed from the DUT CRAM. The golden DUT and test DUT will execute on all the given input vectors. This additional inner loop of the
fault injection methodology is the design execution portion and will only stop if an output error is detected on the DUT output.

As seen in Figure 6.1, after a fault, or multiple faults depending on the fault injection technique, have been injected into the DUT CRAM, the design execution will continue with no further faults injected into the DUT. No additional faults are introduced into the system to allow the injected fault, or faults, to propagate and be present in the system while the DUT operates on all given input test vectors.

After each fault injection, a predetermined delay occurs before the JCM can inject another fault. This time delay is based on the amount of time the DUT will need to operate on all the given input test vectors. This delay is calculated using the clock frequency of the design and the number of input vectors applied to the DUT. The delay must be sufficient enough such that the DUT has time to process all input vectors with the fault present in the system before repairing the current fault and injecting another. This delay allows time for errors caused by CRAMS to propagate to the outputs of the design.

The JCM software implements this time delay after performing each injection. After the JCM has successfully injected a fault in the DUT CRAM, the software will continuously check the system status. This continuous checking occurs during the predetermined delay time before the JCM can repair the previous fault and inject any additional faults. This methodology ensures that each fault, and the resulting design behavior, can be tracked in a fine-grain manner to correlate design failure with the specific injected fault.

6.4 Error Checking and Recovery

After every input test vector, a corresponding output response from both the golden DUT and test DUT is sent to the error detection submodule for comparison. This output response signal corresponds to the 64-bit data signal available across the FMC coupler card. The size of the actual output signal received on the golden device may vary depending on design implementation. These two responses are compared to determine if there is an error in the DUT response, and the system status is updated accordingly by writing to the available BSCAN registers. The JCM continuously monitors the system status by reading the BSCAN registers via JTAG chain access. The JCM continuously logs the system status for every injection and every output response. This continuous
checking by the JCM ensures every error detected by the error detection submodule and logged to the BSCAN registers is logged accurately. This section will discuss the specific steps performed in the fault injection methodology when an error is detected and when no error is detected.

**Error Detected**

If an error is detected, the following steps take place:

1. The golden device sets the failure bit in the system status to 1,
2. The status is written to a BSCAN register,
3. Output responses for both devices corresponding to five clock cycles both before and after the clock cycle when a failure was detected are logged to BSCAN registers,
4. The JCM reads the system status from the BSCAN registers on the golden device through JTAG,
5. The injected bit is classified as sensitive and logged with the system status,
6. The CRAM bit is repaired,
7. The system status is reset and
8. If more faults are to be injected, testing proceeds, otherwise the test terminates.

The output response data logged in step 3 is for post-processing of data in both fault injection and radiation testing (see Appendix A for sample logs). This data is a cycle-by-cycle log of the output response from both the golden design instance and DUT. It contains output responses corresponding to five clock cycles before and after the clock cycle when an error on the DUT output response was detected. This detailed logging provides the ability to examine output responses both before and after the failure occurred. This data can be useful in post-processing fault injection results to closely examine the DUT output on a cycle-by-cycle basis. This data could be as simple as a single response logged for both the golden DUT and DUT being injected with faults.

Additionally, the data logged can correspond to responses from multiple TMR domains within the DUT (see Appendix A for sample logs). When testing an FPGA design with TMR
applied, this logging implementation can provide more insight and understanding of the output response of each TMR domain. This logged output response data could allow the designer to identify specific failure patterns in the DUT or within a single TMR domain. This data log provides additional context and information that will correspond to the system status read from BSCAN registers.

During the fault injection test, the system status must be reset after an error is detected, data logged, and system status written to BSCAN registers. It is also necessary to repair the fault and restore the system status after an error has occurred. In the TURTLE, the method used to reset the status varies depending on the severity of the error. The three main methods to reset the system and system status are soft reset, internal reset, and device reset.

After the JCM has logged all necessary data, the soft reset approach first repairs the injected fault. The JCM then applies a reset signal through the BSCAN register to trigger a reset of the system status in the golden device. This soft reset repairs the fault in the DUT by restoring the CRAM contents to the original value. The fault repair and applied reset signal should eliminate any errors on the DUT output response and restore the 32-bit system status to reflect an error-free operating DUT.

If the soft reset does not restore the system status and eliminate the DUT output error, the JCM will issue an internal reset through the BSCAN registers to the golden device. However, this reset signal will cause the master FSM to transition to the reset state, which will force both the golden DUT and test DUT to perform internal global resets. After applying the reset signals, the FSM will resume normal functionality and ensure both the golden DUT and test DUT are operating synchronously before proceeding with the fault injection test.

Finally, if the error continues, the JCM will perform a device reset. The device reset will reprogram the test FPGA containing the DUT. After reprogramming the test FPGA, the master FSM will aid in synchronizing the golden DUT and test DUT. Once the designs are operating in lockstep, the fault injection test can proceed.

**No Error Detected**

When no error is detected, the designs will operate until all input test vectors from the data generator submodule have been processed. After processing all test vectors, the injected CRAM bit
is repaired by the JCM, and the 32-bit system status is checked to ensure the system is in a known
good state before proceeding with the test. If more faults are to be injected, testing proceeds,
otherwise the test is terminated.

During the fault injection test, the JCM ensures all data is logged, including each injection
and recorded failure with their corresponding system status. These data are written to output files
by the JCM software and can be viewed after the fault injection test for post-processing of data.
Precise error detection and data logging ensure each injection and DUT response is recorded for
further analysis.
CHAPTER 7. FAULT INJECTION CAMPAIGNS

A typical approach for fault injection testing is to perform fault injection first, on an unmitigated design, and then another test on the same design with an applied SEU mitigation technique. This approach is commonly referred to as a single fault injection experiment. Additionally, fault injection tests may seek to compare an unmitigated design to several different SEU mitigation techniques applied to the same design. This approach is commonly referred to as a fault injection campaign [43]. A fault injection campaign may involve multiple experiments that could involve testing several SEU mitigation techniques applied across several designs with different testing parameters. Data collected through fault injection campaigns aid in providing greater confidence in reliability metrics and SEU sensitivity estimates.

The TURTLE platform was used in several different fault injection campaigns. These campaigns used the TURTLE in various ways taking advantage of the fault injection approach explained in the previous chapter. These campaigns tested different designs, SEU mitigation techniques and applied custom fault injection methods as needed. Fault injection is a helpful tool that provides important information on the effectiveness of a given SEU mitigation technique. Additionally, fault injection can provide helpful data about the influence of FPGA architecture and test fixture configuration for SEU sensitivity testing.

There were three goals associated with the presented fault injection campaigns. The first was to estimate the design sensitivity of several benchmark designs with and without applied SEU mitigation techniques. The second was to explore different fault injection approaches such as targeting essential bits or using multi-cell fault injection to emulate upsets events observed in radiation environments. Finally, the third was to compare the SEU mitigation techniques to gain information related to the effectiveness and reliability improvement provided by the applied SEU mitigation techniques. Each of these campaigns will be described in the following sections.
7.1 Benchmark Campaign

The first fault injection campaign targeted four benchmark designs to obtain SEU sensitivity estimates for each unmitigated design. These designs were used as simple baseline benchmarks to perform fault injection and calculate sensitivity with no SEU mitigation techniques applied within the design. This fault injection campaign provides baseline fault injection results used to compare results from the same designs with applied SEU mitigation results. The four different circuits used in this baseline campaign and throughout the other described campaigns for fault injection were the b13, md5, sha3, and aes128:

- **b13** - This design is a simple state machine for a weather balloon. It is part of the ITC’99 benchmark suite [44]. This state machine uses a relatively small amount of FPGA resources. For this work, the b13 is instantiated 256 times within the FPGA and run in parallel. This design inherently uses more FPGA CRAM and logic resources. The output of these circuit copies is routed through a reduction network on the test FPGA before being sent across the FMC coupler card for error detection in the golden FPGA design. If an injected fault causes any of the 256 circuit copies to produce incorrect output. This output reduction network will cause a failure to be detected by the error detection logic in the golden FPGA.

- **md5** - This design is a message-digest algorithm and is used to calculate a checksum to verify data integrity. This circuit takes 512-bit blocks as input to produce a 128-bit hash value. For this thesis work, three copies of the md5 are chained together on a single FPGA.

- **sha3** - This design is the secure hash algorithm 3. This design places two copies of the sha3 chained together in series. In this design, the input stimulus is xor’ed into the design state. The output is then read from the design state.

- **aes128** - This design is the advanced encryption standard. This design operates with a key size of 128 bits and places three different copies chained together in series. With the key size being 128 bits, there are ten transformation rounds in the design.

Each design was used in fault injection testing to estimate the design sensitivity [28]. The baseline implementation and FPGA resource usage details from the Xilinx Artix-7 par of each circuit are shown in Table 7.1. This table presents usage for the following elements:
- **Number of Cells** - This is the number of device cells used in the design implementation on the Artix-7 Xilinx device.

- **Number of Routing Nodes** - This is the number of routing nodes used in routing the design in the Artix-7 Xilinx device.

- **Number of Instances** - This is the total number of design instances that were implemented in available logic resources in the Artix-7 Xilinx device.

### Table 7.1: Base FPGA Utilization of each Benchmark Circuit

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Number of Cells</th>
<th>Number of Routing Nodes</th>
<th>Number of Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>b13</td>
<td>25,388</td>
<td>226,748</td>
<td>256</td>
</tr>
<tr>
<td>md5</td>
<td>65,561</td>
<td>701,011</td>
<td>3</td>
</tr>
<tr>
<td>sha3</td>
<td>20,690</td>
<td>399,059</td>
<td>2</td>
</tr>
<tr>
<td>aes128</td>
<td>56,090</td>
<td>627,345</td>
<td>3</td>
</tr>
</tbody>
</table>

Each design was integrated into the TURTLE platform using the framework and design considerations discussed in Chapter 5. Modifications to the TURTLE software were implemented as needed for each design as part of this campaign. For each baseline design, specific time delays were implemented to allow injected faults to propagate through the system and ensure the implemented FPGA designs operated on all supplied input vectors. The delays implemented for each benchmark design were as follows:

- **b13**: 1.0 ms
- **md5**: 0.5 ms
- **sha3**: 0.5 ms
- **aes128**: 1.5 ms
The delay propagation used for each design was based on design-specific parameters such as operation speed, input test vectors, and output data response time. As this was a baseline campaign, a single TURTLE pond (see section 4.3.2) was used to perform the fault injection campaign. This baseline fault injection campaign successfully demonstrated the scalability of the TURTLE platform. Additionally, this campaign demonstrated and took advantage of the TURTLE’s ability to execute simultaneous fault injection experiments.

With the advantage of executing parallel tests, this baseline campaign performed 6,045,542 injections across all four baseline designs, which are approximately 10% of the CRAM bits. This fault injection campaign took approximately 10.2 hours to complete. Table 7.2 shows the results from this fault injection campaign on the unmitigated versions of each design used in this thesis work. These preliminary results show the design sensitivity for the unmitigated FPGA designs, and results from other fault injection campaigns will be presented in the rest of this chapter. The results from these designs will be analyzed and presented alongside the same FPGA designs but with different applied SEU mitigation techniques tested through fault injection on the TURTLE platform. These results will provide more meaningful conclusions, particularly the number of faults injected on the designs, the methods used in fault injection, and the statistical confidence in the design sensitivity estimates.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Number of Injections</th>
<th>Number of Failures</th>
<th>Design Sensitivity</th>
<th>+95% Confidence</th>
<th>-95% Confidence</th>
<th>Coefficient of Variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>b13</td>
<td>45,542</td>
<td>617</td>
<td>$1.35 \times 10^{-2}$</td>
<td>$1.46 \times 10^{-2}$</td>
<td>$1.25 \times 10^{-2}$</td>
<td>$4.01 \times 10^{-2}$</td>
</tr>
<tr>
<td>md5</td>
<td>2,000,000</td>
<td>129,677</td>
<td>$6.48 \times 10^{-2}$</td>
<td>$6.52 \times 10^{-2}$</td>
<td>$6.45 \times 10^{-2}$</td>
<td>$2.69 \times 10^{-3}$</td>
</tr>
<tr>
<td>sha3</td>
<td>2,000,000</td>
<td>78,376</td>
<td>$3.92 \times 10^{-2}$</td>
<td>$3.95 \times 10^{-2}$</td>
<td>$3.89 \times 10^{-2}$</td>
<td>$3.50 \times 10^{-3}$</td>
</tr>
<tr>
<td>aes128</td>
<td>2,000,000</td>
<td>121,780</td>
<td>$6.09 \times 10^{-2}$</td>
<td>$6.12 \times 10^{-2}$</td>
<td>$6.05 \times 10^{-2}$</td>
<td>$2.78 \times 10^{-3}$</td>
</tr>
</tbody>
</table>
7.2 B13 Exhaustive Test Campaign

Once results had been collected for each benchmark design, another major fault injection campaign conducted an exhaustive test of the b13 benchmark design. The goal of this fault injection campaign was to fully characterize the b13 design. This characterization tested all CRAM bits in the unmitigated and TMR b13 design. The results from this campaign explored how fully triplicating design logic and I/O improved the b13 design sensitivity. The results allowed for further exploration and understanding of how TMR affects design sensitivity. The exhaustive test targeted injecting each user-accessible CRAM bit to determine how the design behavior was affected.

As this campaign sought to inject all the CRAM bits, fault injection tests were executed simultaneously across multiple TURTLE ponds. Each fault injection test targeted a specific section of CRAM addresses. The CRAM addresses were divided across fault injection tests and deployed on a single TURTLE within a TURTLE pond. The software on the JCM that controls fault location selection was modified to target the specific section of CRAM addresses and sequentially inject faults. By modifying the fault injection software, each TURTLE performed fault injection on a distinct section of CRAM addresses with no overlap address overlap between fault injection tests. This approach allowed the exhaustive test to run much quicker than a fault injection platform with a single target device. The flexibility of the TURTLE platform made it easy to modify the fault injection software. The software was modified to implement the needed fault location selection algorithm and divide the CRAM addresses across each test. Additionally, the parallel TURTLE configuration enabled the successful implementation of this campaign.

This fault injection campaign targeted two designs, the fully triplicated and unmitigated b13 design. This fault injection used a total of three TURTLE ponds. The CRAM addresses were partitioned among each pond, as previously discussed. As seen in table 7.3, a total of 118,342,912 injections were performed over the span of 8 days.

This campaign collected the total number of injections, along with the total number of failures detected for each circuit tested. Table 7.3 shows that applying TMR greatly reduced the design sensitivity. One goal for this fault injection campaign was to better characterize which FPGA resource the injected fault affected and caused a failure. Through the large amounts of data collected, the TURTLE was successful in helping to identify key design components that, when upset, could cause design failure.
Table 7.3: B13 Exhaustive Fault Injection Results

<table>
<thead>
<tr>
<th>Mitigation Technique</th>
<th>Number of Injections</th>
<th>Number of Failures</th>
<th>Design Sensitivity</th>
<th>Coefficient of Variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>59,171,456</td>
<td>760,751</td>
<td>$1.29 \times 10^{-2}$</td>
<td>$1.14 \times 10^{-3}$</td>
</tr>
<tr>
<td>Trip-IO</td>
<td>59,171,456</td>
<td>2,223</td>
<td>$3.76 \times 10^{-5}$</td>
<td>$2.12 \times 10^{-2}$</td>
</tr>
</tbody>
</table>

Another important goal of this fault injection campaign was to demonstrate the TURTLE’s ability to perform another form of targeted fault injection. This targeted fault injection consisted of replaying each fault injection that caused a failure in the first test of the fault injection campaign. Through detailed logging on the TURTLE, each injection that caused a failure was injected an additional 10 times. The TURTLE software was able to easily collect and inject all the bits that caused a failure. These additional tests performed an additional 760,974 injections as part of this campaign. The ability to easily replay or re-inject bits on the TURTLE provided further insight and confirmed the bits that caused design failures.

7.3 PCMF Technique Validation Campaign

One major fault injection campaign completed using the TURTLE platform was used to test and validate an SEU mitigation technique used to remove common mode failures (CMF) from a design. This SEU mitigation technique focused on removing CMF through incremental placement of resources in the design [45]. This mitigation technique also aimed at removing CMF due to the failure of clocking signals within a circuit. This technique, named placement CMF or PCMF, uses incremental placing to alter the design placement to eliminate potential sources of failure due to CMF [45]. For the PCMF mitigation technique, the goal of the placement algorithm is to ensure that no device tiles have resources that could lead to a CMF failure.

Multiple TURTLE ponds were used to perform fault injection on several designs and SEU mitigation techniques as part of this campaign. As previously discussed, the TURTLE platform provides flexibility in the fault injection methodology and configuration. For this fault injection campaign, the following test configuration settings were used:
• A single random bit was selected for each injection from the device type 0 frames in the CRAM;

• The same CRAM bit can be chosen multiple times during the campaign;

• The fault is present and propagates through the design for 1 ms after the injection and before the fault is repaired in the CRAM;

• If a failure is detected, the CRAM is scrubbed to repair the fault, and if necessary the device is either reconfigured or power-cycled to reset the design to a good state;

• Each circuit described in section 7.1 was tested with at least 2,000,000 injections or until at least 100 failures were observed;

• In the case when no failures were observed, the circuit was tested with 12,000,000 injections (there are 59,171,456 bits in the CRAM for Xilinx Artix-7 part used in the campaign).

Using the above configuration, the TURTLE performed this fault injection at a rate of about 95 injections per second. With this rate of injection, this fault injection campaign was completed in approximately 28 days.

Several SEU mitigation techniques were tested to compare the improvement of reliability provided from each one. This comparison shows the improvement offered by the PCMF SEU mitigation technique. The total number of injections and observed failures for mitigation technique are shown in Table 7.4. This table also reports each mitigation technique’s sensitivity and the improvement in design sensitivity provided by the mitigation technique over the unmitigated design. Table 7.4 provides insight into which mitigation techniques are more effective in decreasing the design sensitivity.

These results present the following important metrics that are collected during fault injection:

• **Mitigation Technique** - This is the SEU mitigation technique applied to the FPGA design. These techniques were applied to several FPGA designs during the fault injection campaigns (see Appendix B for sample TCL scripts for implementing these mitigation techniques).
• **Number Injections** - This is the number of faults injected into the device CRAM. In Table 7.4, this number is the total number of injections across all the benchmark circuits for each mitigation technique;

• **Number of Failures** - This is the number of failures observed during fault injection. For this campaign, a single injected fault could result in at most one failure. For Table 7.4, these are the total number of failures across all benchmark circuits;

• **Design Sensitivity** - The number is the probability that a random bit flip will cause a failure. This is the number of failures divided by the number of injections. Because this table shows the geomean bit sensitivity, the bit sensitivity for each mitigation technique is the geomean average of the bit sensitivities for each benchmark circuit.

• **Improvement** - The design sensitivity improvement in relation to the unmitigated design. This metric is calculated by dividing the unmitigated design sensitivity by the mitigation technique’s design sensitivity.

• **Coefficient of Variation** - The coefficient of variation measures the dispersion of distribution— for fault injection results, a large coefficient of variation results in larger confidence intervals meaning that the range of the actual design sensitivity from the estimated sensitivity is larger, conveying less confidence in the estimated design sensitivity. A large number of failure events need to be observed for a smaller coefficient of variation, which directly relates to the number of injections performed.

This fault injection campaign was successful in showing that the PCMF SEU mitigation technique did decrease the design sensitivity to SEUs. Since the fault injection campaign injected only single faults into the CRAM, it can be stated that the mitigation technique decreases the design sensitivity to single events in the CRAM.

The TURTLE platform was vital in providing large amounts of fault injection to provide more accurate estimations of the design sensitivities with various SEU mitigation techniques applied to different designs. These results shown in Table 7.4 provided promising results for the PCMF SEU mitigation technique to be further tested in radiation testing [28].
Table 7.4: Fault Injection Geomean Results for PCMF Validation Campaign

<table>
<thead>
<tr>
<th>Mitigation Technique</th>
<th>Number of Injections</th>
<th>Number of Failures</th>
<th>Design Sensitivity</th>
<th>Improvement</th>
<th>Coefficient of Variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>6,045,542</td>
<td>330,450</td>
<td>3.80 x 10^-2</td>
<td>1 x</td>
<td>2.43 x 10^-3</td>
</tr>
<tr>
<td>Common-IO (1-Voter)</td>
<td>6,037,318</td>
<td>16,960</td>
<td>1.54 x 10^-3</td>
<td>24.8 x</td>
<td>1.40 x 10^-2</td>
</tr>
<tr>
<td>Common-IO (3-Voter)</td>
<td>6,132,724</td>
<td>3,367</td>
<td>3.16 x 10^-4</td>
<td>121 x</td>
<td>2.99 x 10^-2</td>
</tr>
<tr>
<td>Split-IO</td>
<td>8,000,000</td>
<td>5,446</td>
<td>8.39 x 10^-5</td>
<td>453 x</td>
<td>1.10 x 10^-1</td>
</tr>
<tr>
<td>Split-clock</td>
<td>6,538,320</td>
<td>778</td>
<td>1.13 x 10^-4</td>
<td>337 x</td>
<td>3.78 x 10^-2</td>
</tr>
<tr>
<td>Split-clock PCMF</td>
<td>6,336,939</td>
<td>486</td>
<td>8.45 x 10^-5</td>
<td>451 x</td>
<td>4.12 x 10^-2</td>
</tr>
<tr>
<td>Early Split</td>
<td>9,255,262</td>
<td>313</td>
<td>2.27 x 10^-5</td>
<td>1,675 x</td>
<td>8.42 x 10^-2</td>
</tr>
<tr>
<td>Early Split PCMF</td>
<td>18,000,000</td>
<td>377</td>
<td>1.63 x 10^-5</td>
<td>2,340 x</td>
<td>6.62 x 10^-2</td>
</tr>
<tr>
<td>Trip-IO (1-Voter)</td>
<td>24,000,000</td>
<td>50,840</td>
<td>4.56 x 10^-4</td>
<td>83.4 x</td>
<td>2.06 x 10^-2</td>
</tr>
<tr>
<td>Trip-IO (3-Voter)</td>
<td>28,000,000</td>
<td>223</td>
<td>2.05 x 10^-6</td>
<td>18,576 x</td>
<td>2.60 x 10^-1</td>
</tr>
<tr>
<td>Trip-IO PCMF</td>
<td>28,000,000</td>
<td>49</td>
<td>8.18 x 10^-7</td>
<td>46,511 x</td>
<td>3.06 x 10^-1</td>
</tr>
</tbody>
</table>

7.4 MCU Fault Injection Campaign

This section will highlight some of the work done with MCU fault injection [46]. This section shows how the TURTLE can implement different methods of fault injection algorithms and correlate radiation test data to fault injection results.

In addition to collecting large amounts of fault injection data through performing parallel fault injection tests, the TURTLE has highly-customizable software that allows for different fault injection algorithms to be implemented. One fault injection campaign focused on multi-cell upsets that cause failures in FPGA designs.

Performing fault injection of MCUs involves additional challenges. First, an MCU involves injecting faults in a defined pattern based on upset data collected through radiation tests [9]. This pattern defines the MCU shape used to perform the fault selection location as shown in Figure 7.1. Injecting an MCU shape requires modifications to the fault selection to select specific patterns of bits to be injected as the MCU shape in the CRAM.

Second, it might not be possible to inject the desired MCU shape in the selected location. It may not be possible due to an invalid logical bit location that prevents the user from accessing the bit or CRAM address. For example, in attempting to inject an MCU shape from Figure 7.1, if
the starting bit is randomly selected, the remaining bits to inject as part of the MCU shape may be invalid locations that are out of range of valid CRAM addresses. This obstacle makes it difficult to select any random bit as the first bit in the MCU shape as it may result in invalid bit locations for the remaining MCU shape. This obstacle is an added constraint that must be handled during the fault location selection process.

Third, even though the user can attempt to inject any MCU shape in random bits of the device, the injected faults could violate the nature of MCUs. This violation may occur because it is not possible to determine if the injected MCU shape could be a product of a single charged particle [9]. Potentially, the selected bits are not physically close to each other. To overcome this challenge, it is necessary to perform statistical analysis on the MCUs’ locations and restrict the valid location to inject an MCU [9].

Injecting MCUs presents different challenges than single-bit fault injection. To successfully inject an MCU and model the behavior of upsets observed in radiation beam tests, the selected CRAM addresses to inject must be constrained to certain locations within the device CRAM. An additional consideration when performing MCU fault injection requires that all bits injected as part of the MCU shape are simultaneously present in the CRAM. This approach differs from single-bit fault injection as a single injected fault is present in the CRAM at a given time before the fault is repaired. For MCU fault injection, all injected faults are allowed to propagate in the system before repairing all injected bits as part of the MCU shape. This particular aspect of injecting MCUs requires a change to the normal fault injection flow.
To inject an MCU, each bit that composes the MCU shape must be injected with minimal delay between each injection. The fault injection system keeps track of all the injected bits to repair each injection when needed. After injection of the MCU shape, the fault injection flow follows the baseline fault injection methodology. The system status is continuously monitored, and any detected failures are reported and logged. The fault injection algorithm then needs to repair all faults that were injected as part of the MCU to avoid the accumulation of upsets in the design. This fault injection campaign was comprised of two experiments discussed in the following subsections that were completed in approximately 5 days.

7.4.1 Single-bit vs Multi-cell Injection Experiment

The first performed an additional MCU fault injection on the B13 design. Table 7.5 shows the different SEU mitigation techniques applied to the B13 during this experiment. The goal of injecting MCUs in this experiment was to show that some failures will only occur in a design if multiple upsets are present in the system during operation, i.e., if an MCU has occurred.

To achieve this goal, first, each design was injected with single-bit upsets collected from radiation testing. The purpose of injecting single-bit upsets was to confirm which single-bit upsets could cause a DUT failure. Each single-bit injection that caused a design failure was classified and recorded. Table 7.5 shows the total number of failures caused by single-bit injections for each design.

After injecting single-bit injections, each design was injected with corresponding MCU shapes reconstructed from upset data collected from radiation beam tests. This experiment showed an additional number of failures only occur in the presence of an MCU. It is interesting to note that the percentage of additional design failures seen during MCU fault injection surpassed 100% in three of the tested designs. This experiment showed that injecting MCUs could cause additional failures not caused by single-bit injections.

7.4.2 Additional MCU Injection Experiment

The second experiment in this fault injection campaign performed additional MCU fault injection on two specific designs. These designs were the PCMF and Striped TMR designs. These
Table 7.5: Summary of results for single-bit and MCU fault injection

<table>
<thead>
<tr>
<th></th>
<th>Failures</th>
<th>Additional MCU Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single-bit</td>
<td>MCU</td>
</tr>
<tr>
<td>Unmitigated</td>
<td>893</td>
<td>73</td>
</tr>
<tr>
<td>Common-IO (3-Voter)</td>
<td>50</td>
<td>11</td>
</tr>
<tr>
<td>Split-clock</td>
<td>29</td>
<td>7</td>
</tr>
<tr>
<td>Trip-IO (3-Voter)</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>Early Split PCMF</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Early Split TMR</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>PCMF</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Striped TMR</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Designs have two SEU mitigation techniques applied that implement TMR and additional placement and routing constraints to mitigate failures from SEUs. These two designs did not experience any failures in the first fault injection experiment of this campaign.

The additional fault injection performed on these two designs included injecting additional MCU shapes reconstructed from radiation test data not previously injected into these two specific designs during the first experiment. The goal of this experiment was to demonstrate the need to use MCU or other complex fault injection methodologies to test high reliable SEU mitigation techniques. As neither of these designs experienced failures in the previous experiment in this campaign, a greater amount of MCUs injections were performed to determine if any design failures would occur from MCUs in either design.

Table 7.6 shows the results of this second experiment. The number of failures is divided into two columns. The first column shows the number of failures that occurred in the design during radiation testing. The second column shows the number of failures that occurred during the additional MCU fault injection. It is interesting to note that the number of failures seen in fault injection increased for both designs from the number of failures seen during radiation beam testing.

For this campaign, over 213,000 MCUs corresponding to 713,000 upsets were injected between both experiments. This campaign benefited from the versatile TURTLE platform by implementing the MCU fault location selection algorithm. Additionally, the TURTLE was used to
Table 7.6: Results for validation of MCU radiation beam testing

<table>
<thead>
<tr>
<th>Design</th>
<th>Failures</th>
<th>Increase in Failures</th>
<th>Total Upsets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Beam test</td>
<td>Fault Injection</td>
<td></td>
</tr>
<tr>
<td>Striped TMR</td>
<td>37</td>
<td>38</td>
<td>2.7%</td>
</tr>
<tr>
<td>PCMF</td>
<td>2</td>
<td>9</td>
<td>350%</td>
</tr>
</tbody>
</table>

correlate radiation beam test data through fault injection tests by replaying upsets observed during the radiation test and confirming observed failures.

7.5 Fault Injection Campaign Comparison

Each fault injection campaign provided further insight into each design and any applied SEU mitigation technique. The benchmark campaign provided baseline results for the unmitigated version of the designs used throughout the various campaigns discussed in this chapter. The benchmark campaign also showed that custom designs could be integrated into the TURTLE platform following the design preparation guidelines discussed in Chapter 5.

The b13 exhaustive fault injection campaign highlighted several abilities of the TURTLE platform. First, it highlighted the ability to run fault injection tests in parallel on three TURTLE ponds. The ability to execute simultaneous fault injection tests increased the number of injections able to be performed in a given time period compared to that of a single TURTLE platform. Most notably, it highlighted the TURTLE’s ability to target and inject large amounts of CRAM bits by partitioning the workload across the TURTLE ponds and proved the JCM’s ability to control and monitor several tests operating in parallel.

The PCMF validation campaign further highlighted the usefulness of the TURTLE framework used to integrate custom designs. This campaign applied a total of 11 different mitigation techniques to each baseline design. This campaign played an essential role in validating the PCMF SEU mitigation technique by collecting large amounts of data across multiple FPGA designs with this technique applied. The TURTLE platform was essential in providing the needed fault injection testing to develop and validate the PCMF SEU mitigation technique.

Finally, the MCU testing campaign highlighted the flexibility of the TURTLE platform. The fault injection selection methodology was modified to implement the injection of MCU shapes.
The ability to adapt the fault location selection methodology was a great benefit in this campaign provided through the TURTLE platform. Additionally, this campaign took advantage of the TURTLE’s ability to replay and re-inject faults from previously collected upset data at radiation beam tests as well as from previous fault injection tests.
CHAPTER 8. CONCLUSION

This thesis presents a framework for performing fault injection testing on SRAM-based FPGAs. This framework is built around a golden/DUT model described in section 4.1. This work implements the golden/DUT model using two separate FPGA devices, one FPGA implementing the golden design instance and the other FPGA implementing the DUT exposed to faults through fault injection. The TURTLE platform provides both the necessary hardware to perform fault injection and a framework to integrate custom designs into the framework. This framework allows for a variety of FPGA designs to be implemented and tested through fault injection.

The effectiveness of the TURTLE platform was demonstrated through multiple fault injection campaigns. Several fault injection campaigns were conducted on the TURTLE platform to estimate the SEU sensitivity of FPGA designs and estimate the effectiveness of several SEU mitigation techniques. This thesis work and experiments accomplished three main goals. First, statistically significant amounts of data were collected to estimate the SEU sensitivity of FPGA designs and accurately estimate the effectiveness of several SEU mitigation techniques. Second, the TURTLE framework and architecture enabled several FPGA designs to be implemented and tested through fault injection. The fault injection platform also provided the ability to test several SEU mitigation techniques across several FPGA designs to compare the improvement provided by each SEU mitigation technique and determine which provided the most favorable results. Finally, the TURTLE software provided the ability to implement highly customizable fault injection campaigns to perform random, targeted, and other types of fault injection methodologies.

The TURTLE fault injection platform presented in this paper was built using relatively low-cost hardware. The fault injection system presented in this work successfully tested and implemented several FPGA designs and SEU mitigation techniques on a Xilinx Artix-7 FPGA. A golden design was operated in lockstep to the DUT to detect and log functional errors. Additionally,
through the parallel TURTLE configuration, the fault injection campaigns performed statistically significant amounts of injections leading to better confidence in design sensitivity measurements.

More than 600 million injections were performed across all executed fault injection campaigns. These fault injection campaigns collected significant data to test several FPGA designs and validate several SEU mitigation techniques. The data obtained from these campaigns were used to calculate accurate design sensitivities for FPGA designs and calculate the coefficient of variation for the sensitivity estimates to obtain better confidence in the sensitivity estimates. Additionally, the data was used to calculate the effectiveness of the applied SEU mitigation techniques by comparing the improvement in design sensitivity across the fault injection campaigns. Both unmitigated and mitigated versions of the benchmark designs mentioned in Chapter 7 were tested through fault injection. The TURTLE also provided key data in identifying areas where the tested SEU mitigation techniques needed improvement.

The TURTLE fault injection platform developed and used in this thesis work was vital for SEU sensitivity estimation and SEU mitigation techniques testing for several research efforts [45], [28], [47], [48]. This work helped progress fault injection testing and is important for future improvements to fault injection testing.

8.1 Benefits and Drawbacks

While the approach presented in this thesis used more FPGA resources, it also provided the ability to create flexible fault injection tests and to test unique SEU mitigation techniques by triplicating I/O into the design or other configurations. This approach also separates the essential test fixtures and logic from the target DUT subject to fault injection testing. The TURTLE platform also took advantage of using two FPGA devices by creating an HDL framework that simplifies design implementation with reusable modules across different designs as discussed in Chapter 5. The major driving goals for developing the TURTLE was to create a low-cost fault injection platform that would allow for highly customizable tests and implement a scalable approach to collect statistically significant data to validate and test highly reliable SEU mitigation techniques.

One drawback of using multiple FPGAs in the fault injection platform developed in this thesis is that it requires more resources than a single FPGA platform. However, this added FPGA provides the certainty that functional logic used to operate the golden design and other error de-
tection logic operates error-free. One other drawback is that because data is compared between two FPGA designs operating in lockstep on separate devices, there is an added delay in the fault injection platform required for the transmission and reception of this data to the golden FPGA. Additionally, the TURTLE uses an external device, the JCM, to control and implemented the fault injection methodology. While this also slightly slows down the rate of fault injection, it provides flexibility in performing fault injection that other platforms do not provide, such as MCU fault injection. The TURTLE platform also provides scalability to implement parallel fault injection tests. The parallelized TURTLE platform increases the number of injections that can be performed, all while using relatively low-cost hardware.

As fault injection techniques and platforms continue to improve, this platform can be used to improve fault injection rates, improve fault injection methodologies to better mimic radiation beam test behavior, and improve the confidence in developed SEU mitigation techniques for a variety of designs. Additionally, this research will help further develop and improve fault injection techniques, provide a platform to collect statistically significant data to analyze FPGA design sensitivity estimates and SEU mitigation techniques, and provide a flexible platform for fault injection experiments.

8.2 Future Work

The framework and results presented in this thesis built upon work presented by previously developed platforms and added meaningful improvements that make the TURTLE fault injection platform a valuable tool in testing FPGA designs and validating SEU mitigation techniques. However, there are improvements that can be made to more effectively test FPGA designs and SEU mitigation techniques through fault injection. The experiments presented in this thesis work were performed on relatively simple FPGA designs; however, commercial high-performance systems or FPGA designs deployed for space-based applications typically result in more complex FPGA designs. These designs would have more complex failure modes that may not have been encountered in this work. Additionally, more complex designs may require additional support modules or design considerations that extend beyond the ones presented in this work. A more featureful framework may provide more support in integrating more complex FPGA designs and may allow for more accurate testing of complex high-end commercial FPGA designs.
The TURTLE platform would benefit from targeting and testing more complex FPGA designs. To integrate more complex FPGA designs for fault injection, the TURTLE could benefit from additional modifications to the platform framework that could involve changes to the various submodules. Additionally, the framework could benefit from modifications of how input and output are sent and received from both FPGA designs. These modifications could include an upgrade in the I/O interface, such as Ethernet or other I/O interfaces that could allow for more complex data packets to be exchanged between devices.

Finally, this work specifically targeted FPGA designs and SEU mitigation techniques developed on Xilinx 7-Series devices. Future work could translate the TURTLE framework to other FPGA devices. The work and experiments presented in this thesis could be applied to other FPGA devices to obtain reliability metrics, which would help provide data to understand how developed SEU mitigation techniques may improve design reliability on different FPGA devices and architectures. Different devices, such as Virtex-4, 5, 6, and UltraScale, would benefit from a fault injection platform like the TURTLE platform. While some platform changes would be required, such as coupling device I/O through different interfaces, the general TURTLE framework could be applied to different devices and used to perform fault injection tests. The required changes to implement the TURTLE platform on other FPGA devices would require a relatively low amount of framework modifications.
REFERENCES


92


APPENDIX A. FAULT INJECTION TESTING LOGS

Fault injection data was collected following the approach outlined in Chapter 6. This appendix includes samples from the data logs collected during fault injection testing. This data was used to calculate and analyze the SEU sensitivity estimate for each FPGA design tested. The presented logs report the injected fault in the format of \(<frame>\) \(<word>\) and \(<bit>\).
A.1 Sequential Fault Injection Log

The following is a sample log from an exhaustive sequential fault injection test run on the B13 unmitigated design. This sample log implements the most basic approach of logging each individual injection, and reporting if a failure occurred.
A.2 Sequential Replay Fault Injection Log

The following is a sample log from replaying identified bits that caused failure during an exhaustive sequential fault injection test run on the B13 unmitigated design. This sample log shows how each bit, already identified to cause failure, can easily be replayed on the TURTLE.

```
[INJE] 0x00000201 28 25
[FAIL] 0x00000201 28 25
[INJE] 0x00000201 28 25
[FAIL] 0x00000201 28 25
[INJE] 0x00000201 28 25
[FAIL] 0x00000201 28 25
[INJE] 0x00000400 57 22
[FAIL] 0x00000400 57 22
[INJE] 0x00000400 57 22
[FAIL] 0x00000400 57 22
[INJE] 0x00000400 57 22
[FAIL] 0x00000400 57 22
[INJE] 0x00000580 20 22
[FAIL] 0x00000580 20 22
[INJE] 0x00000580 20 22
[FAIL] 0x00000580 20 22
[INJE] 0x00000580 20 22
[FAIL] 0x00000580 20 22
[INJE] 0x00000581 32 22
[FAIL] 0x00000581 32 22
[INJE] 0x00000581 32 22
[FAIL] 0x00000581 32 22
[INJE] 0x00000581 32 22
[FAIL] 0x00000581 32 22
[INJE] 0x00000581 32 22
[FAIL] 0x00000581 32 22
[INJE] 0x00000600 44 22
[FAIL] 0x00000600 44 22
[INJE] 0x00000600 44 22
[FAIL] 0x00000600 44 22
[INJE] 0x00000600 44 22
[FAIL] 0x00000600 44 22
[FAIL] 0x00000600 44 22
[FAIL] 0x00000600 44 22
```
A.3 Random Fault Injection Sample Log

The following is a sample log from a random fault injection test run on the B13 unmitigated design. This sample log shows failure detection, and the recovery method used to remove the error from the system. It also reports the running total injections, failures and design sensitivity.

```
[INJE] 0x00422a1b 69 16
[INJE] 0x00000319 99 6
[INJE] 0x00420981 7 28
[INJE] 0x00022c98 49 3
[INJE] 0x00022e1a 79 2
[INJE] 0x00440606 22 5
[INJE] 0x0000261b 62 1
[INJE] 0x00401b99 85 10
[INJE] 0x00402397 63 27
[FAIL] FAILURE DETECTED 0x00402397 63 27
RECOVERY_MODE: SCRUB and ERROR_CLEAR
[STAT] Failures=>43 Injections=>3169 Sensitivity=>0.0136%
[INJE] 0x00440a97 28 26
[INJE] 0x0040161f 12 30
[INJE] 0x00002b19 23 16
[INJE] 0x00421e02 19 25
[INJE] 0x00022b15 87 15
[INJE] 0x0042338a 97 18
[INJE] 0x0040203 47 10
[INJE] 0x00440e01 8 26
[INJE] 0x00422b9c 18 23
[INJE] 0x00001086 25 11
[INJE] 0x00420e87 73 2
[INJE] 0x0044269f 66 21
[INJE] 0x00422b86 61 26
[INJE] 0x004207a3 46 31
[INJE] 0x0040240a 91 17
[FAIL] FAILURE DETECTED 0x0040240a 91 17
RECOVERY_MODE: ERROR_CLEAR
[STAT] Failures=>44 Injections=>3184 Sensitivity=>0.0138%
[INJE] 0x00402f81 12 31
[FAIL] FAILURE DETECTED 0x00402f81 12 31
RECOVERY_MODE: ERROR_CLEAR
[STAT] Failures=>45 Injections=>3185 Sensitivity=>0.0141%
[INJE] 0x00420f15 25 23
[INJE] 0x00402e98 83 24
[INJE] 0x00000e8d 2 26
[INJE] 0x004207a3 46 31
[INJE] 0x0000290e 40 17
```
A.4 MCU Fault Injection Log

The following is a sample log from a MCU fault injection test run on the B13 unmitigated design. This sample log shows how an MCU consists of injecting multiple bits, checking for failure, and repairing all bits injected as part of the MCU as discussed in section 6.2.

[INFO] Creating tables for log storage
[INFO] Creating campaign #543
Calibrating device index => 0
Calibrating device...
Calibration successful!
Extra shifts: 0
Clock rate: 25 MHz
ID CODE => 0x13636093
Calibrating device index => 1
Calibrating device...
Calibration successful!
Extra shifts: 0
Clock rate: 25 MHz
ID CODE => 0x13636093
[INFO] Injecting MCU 1
[10:04:31] Injected 1-bit fault at address 0x0228A2, word 24, bit 28
[INJE] 0x000228A2 24 28
[10:04:31] Injected 1-bit fault at address 0x0228A3, word 24, bit 27
[INJE] 0x000228A3 24 27
[SCRB] 0x000228A2:24=>0x18000000
[SCRB] 0x000228A3:24=>0x30000000
[INFO] Injecting MCU 2
[10:04:31] Injected 1-bit fault at address 0x0228E8C, word 8, bit 31
[INJE] 0x000228E8C 8 31
[10:04:31] Injected 1-bit fault at address 0x0228E8D, word 8, bit 29
[INJE] 0x000228E8D 8 29
[SCRB] 0x000228E8C:8=>0x60000000
[SCRB] 0x000228E8D:8=>0xC0000000
[INFO] Injecting MCU 3
[10:04:31] Injected 1-bit fault at address 0x440F86, word 45, bit 10
[INJE] 0x00440F86 45 10
[10:04:31] Injected 1-bit fault at address 0x440F87, word 45, bit 11
[INJE] 0x00440F87 45 11
[SCRB] 0x00440F86:45=>0x00000080
[SCRB] 0x00440F87:45=>0x00000040

A.4 MCU Fault Injection Log

The following is a sample log from a MCU fault injection test run on the B13 unmitigated design. This sample log shows how an MCU consists of injecting multiple bits, checking for failure, and repairing all bits injected as part of the MCU as discussed in section 6.2.
A.4.1 MCU Fault Injection TMR Domain Results

The following is a sample log from a MCU fault injection test run on the B13 unmitigated design. This sample log particularly shows how data from each TMR domain is reported when a failure is detected.

Injecting MCU 370
[10:05:15] Injected 1-bit fault at address 0x00038C, word 34, bit 2
[INJE] 0x0000038c 34 2
[10:05:15] Injected 1-bit fault at address 0x00038C, word 34, bit 20
[INJE] 0x0000038c 34 20
[10:05:15] Injected 1-bit fault at address 0x00038D, word 34, bit 22
[INJE] 0x0000038d 34 22
[10:05:15] Injected 1-bit fault at address 0x00038D, word 34, bit 21
[INJE] 0x0000038d 34 21
[10:05:15] Injected 1-bit fault at address 0x00038D, word 34, bit 20
[INJE] 0x0000038d 34 20
[10:05:15] Injected 1-bit fault at address 0x00038E, word 34, bit 23
[INJE] 0x0000038e 34 23
[10:05:15] Injected 1-bit fault at address 0x00038E, word 34, bit 24
[INJE] 0x0000038e 34 24
[10:05:15] Injected 1-bit fault at address 0x00038E, word 34, bit 22
[INJE] 0x0000038e 34 22
[10:05:15] Injected 1-bit fault at address 0x00038F, word 34, bit 23
[INJE] 0x0000038f 34 23
[10:05:15] Injected 1-bit fault at address 0x00038F, word 34, bit 22
[INJE] 0x0000038f 34 22
[10:05:15] Injected 1-bit fault at address 0x00038F, word 34, bit 24
[INJE] 0x0000038f 34 24
[10:05:15] Injected 1-bit fault at address 0x000390, word 34, bit 24
[INJE] 0x00000390 34 24

[FAIL] FAILURE DETECTED 0x00000390 34 24
[STATUS] 0x43434cb

[DATA] M=>00000091 S0=>00000091 S1=>00000091 S2=>00000091
[DATA] M=>00000091 S0=>00000091 S1=>00000091 S2=>00000091
[DATA] M=>00000095 S0=>00000095 S1=>00000095 S2=>00000095
[DATA] M=>00000095 S0=>00000095 S1=>00000095 S2=>00000095
[DATA] M=>000000295 S0=>00000168 S1=>00000168 S2=>00000168
[DATA] M=>000000391 S0=>0000006c S1=>0000006c S2=>0000006c
[DATA] M=>000000364 S0=>000000364 S1=>000000364 S2=>000000364
[DATA] M=>000000364 S0=>000000364 S1=>000000364 S2=>000000364
[SCR] 0x00000390:34=>0x01000000
[SCR] 0x0000038f:34=>0x00000000
[SCR] 0x0000038e:34=>0x01000000
[SCR] 0x0000038d:34=>0x00070000
[SCR] 0x0000038c:34=>0x00070000

[STAT] Status=>[88.61%] Failures=>9 Injections=>46000 Sensitivity=>0.0196%
APPENDIX B. FAULT INJECTION SAMPLE CODE, DESIGN VARIATION, AND I/O CONSTRAINT SCRIPTS

This appendix contains various sample scripts and code used to generate FPGA designs such as variations of TMR, and to perform fault injection tests from the JCM as described in section 4.2.4.
This example software controls aspects of the fault injection flow discussed in Chapter 6.

```python
import random
import signal
import time
import select
from typing import Iterable, List, Tuple
import sys
from swig import jcm
from swig.tile import Logger
from swig.tile.Scrubber import FrameBasedScrubber, ScrubbedBits
from itertools import combinations

class TurtleFaultInjection:
    def __init__(self, test_name, description, master_bitfile, slave_bitfile, jtagport, config_file, verbose):
        self.turtle = TurtleFaultInjection(test_name, description, master_bitfile, slave_bitfile, jtagport, config_file, verbose)

    def random_bits(self, values, num, repetitions):
        self.turtle.random_bits(values, num, repetitions)

    def test(self, terminnate):
        if not terminnate:
            self.turtle.random_bits(values, num, repetitions)
        else:
            print("Press CTRL+C to force immediate termination")
```
def get_jtagCM(self) -> jcm.JTAGCMChainManager:
    return jcm.JCMJTAGChainManager()

def set_active_device(self, index: int):
    self.chainManager.setActiveDeviceIndex(index)
    self.JTAGConfigInterface.getHugInOutCommands().setExtraShiftMask(self.chainManager.getActiveShiftAt(index))

    def calibrate_device(self, config_file: str, port: int, speed: int, debug, device_id):
        self.set_active_device(device_id)
        if self.verbose:
            print("Calibrating device index ": self.chainManager.getActiveDeviceIndex())
        self.calibrate_calibrateDevice(config_file, port, speed, False, False)
        configFileIdCode = self.chainManager.getActiveDevice(self.getFileCode)
        jtagIdCode = self.JTAGConfigInterface.readJtagIdCode()
        self.xilinx = jcm.XilinxReadConfigRegisterCommandSequence.to_XilinxCommandSequence(self.xilinxIdCode)
        configId = jcm.u32p(self.JTAGConfigInterface.readConfiguration(self.xcm8))

        def inject_bit(self, bit: Tuple[int, int, int]) -> bool:
            self.set_active_device(self.slave_device_id)
            return self fsm.injectFault(bit)

        def propagate(self) -> None:
            self.time.sleep(self.propagation_time)
            self.get_bscan_status(self.master_id)
            for _ in range(self.status_attempts):
                status_ptr = self.JTAGConfigInterface.readBscan(self.STATUSBSCAN, self.STATUSBSCAN_LENGTH)
                if self.verbose:
                    print(self.STATUS, status_ptr)
                self.delete_u32p(status_ptr)
                status = status_ptr & self.BSCAN_CONSTANT
                if status:
                    status &= self.BSCAN_CONSTANT
                    return

        def check_device_status(self, status, int) -> bool:
            return not (status & self.BSCAN_JTAGERRORMASK)

        def scrub(self, frame) -> List[ScrubbableBits]:
            self.set_active_device(self.slave_device_id)
            frameInfo = ConfigBitInfo(frame)
            if frameInfo.minor in [30, 31]:
                frameInfo.minor = 30
            return self.fbs.short_readback(frameInfo, frame, 6, True)
self.clear_error()  
self.propagate()  
new_system_status_raw = self.get_bscan_status()  
if self.check_device_status(new_system_status_raw)  
return self.log(SCRUB_AND_ERRORCLEAR, scrubbed_bits)  

# System Reset Recovery  
self.reset_device()  
self.propagate()  
new_system_status_raw = self.get_bscan_status()  
if self.check_device_status(new_system_status_raw):  
return self.log(SYSTEM_RESET, scrubbed_bits)  

# Full Reconfigure Recovery  
sel self.full_reconfigure_recovery()  
return self.log(FULL_RECONFIGURE, scrubbed_bits)  

def init_test(self) -> int:  
cmp_id = self.log.create_campaign(test_name=self.test_name,  
description=self.description)  
self.calibrate_device(config_file=self.jtagport, self,  
CALIBRATION_SPEED=False, self.master_idx)  
self.calibrate_device(config_file=self.jtagport, self,  
CALIBRATION_SPEED=True, self.master_idx)  
self.set_active_device(selfslave_idx)  
sel self.full_device_configure(selfslave_idx)  
sel bfs = FrameBasedScrubber(XilinxScrubber=self, scrubberUtility=  
XilinxPGAINfo=self.xilinxPGAINfo,  
cmdGenerator=self,  
xilinxReadback=True, self.xilinxFaultInJect=  
configInterface=self,  
bscanInJect=  
configInterface=,  
pgainInfo=0, self, bscanInJect=  
initReadback=True, self.jtagport, bscan=b  
golden_data_filename="/tmp/slid.data")  
sel self.full_device_configure(selfmaster_idx)  
device_status = False  
for i in range(self.reset_attempts):  
sel self.reset_device()  
status = self.get_bscan_status()  
device_status = self.check_device_status(status)  
if device_status:  
  break  
  
if not device_status:  
  print(self.ERRMESSAGE, format("Failed to reset device into a good  
state"))  
  exit(1)  

return cmp_id  

def run_test_experiment(self, bits: Iterable[Tuple[int, int, int]]) -> None:  
cmp_id = self.init_test()  

def in range(5):  
bfs = 10 ** i  
print("([port {]) Speed test on rapid fault injection of {} bits: ",  
format(self.jtagport, bits, end="")  
startTime = time.time()  
for j in range(bfs):  
sel self.inj_bit(0x00442F96, 17, 16)  
sel self.scrub(0x00442F96)  
stopTime = time.time()  
totalTime = stopTime - startTime  
print("[] total: {} / fault: format(totalTime, totalTime/bits)")  

def random_bits(self, values: int, repetitions: int) -> Iterable[Tuple[int,  
int, int]]:  
num_frames = self.xilinxPGAINfo.getNumLogicFrames()  
num_words_per_frame = self.xilinxPGAINfo.getWordsPerFrame()  
frame_addresses = jmc u32a.frompointer(self.xilinxPGAINfo.  
getFracStructure().getFracArray())  
for current_value in range(values):  
  status = float(current_value) / float(values)  
  if self.total_injections > 0:  
    sensitivity_percent = float(self.total_failures) / float(self.total_injections)  
    else:  
      sensitivity_percent = 0.0  
  self.log.print(self.STATUS_MESSAGE.format(status_percent, self,  
  total_failures, self.total_injections, sensitivity_percent))  
frame = random.randint(0, num_frames)  
word = random.randint(0, num_words_per_frame)  
bit = random.randint(0, 31)  
address = frame_addresses[frame]  
if address == 0xFFFF0000:  
  continue  

for i in range(repetitions):  
yield(address, word, bit)  

def run_test(self, bits: Iterable[Tuple[int, int, int]]) -> None:  
self.init_test()  
for bit in bits:  
  if self.configRad = bit[0]:  
    keypress = select.select(self.inputSrc, [ ] , [ ] , 0)[0]  
p =  
  while keypress:  
    for src in keypress:  
      line = src.readline()  
      if not line:  
        self.inputSrc.remove(src)  
    else:  
      if line == "c":  
        print("Test paused")  
        print(self.STATUS_MESSAGE.format(self.total_failures,  
        self.total_injections, self.failed_injections))  
        self.log.close()  
        self.log.write("[STAT] Injection campaign complete")  
        self.master_log.write("[STAT] Injection campaign complete")  
        self slave_log.close()  
        self master_log.close()  
        exit(0)  
      if p == 'c':  
        break  
    else:  
      self slave_log.write("[STAT] Injection campaign complete")  
      self master_log.write("[STAT] Injection campaign complete")  
      self slave_log.close()  
      self master_log.close()  
      exit(0)  
    if p == 'c':  
      break  
  if not self.inject_bit(bit):  
    self.failed_injections += 1  
    continue  
  if self.verbose:  
    print(self.INJECT_MESSAGE.format(bit))  
    self.total_injections += 1  
    self.propagate()  
  self.system_status = self.get_bscan_status()  
  if there is no design FAILURE  
  if self.check_device_status(system_status, raw):  
    scrubbed_bits = self.scrub(bit[0])
B.3 TMR Design Voter Variation

An example TCL script used in Vivado to modify an FPGA design by removing specific TMR voters from the design.

```tcl
set voters_all [sort dictionary [get_cells hierarchical regex [-TMR_VOTER, 012]]]
for {set i 0} {($i < [expr $voter_count / 3])} {incr i} {
  set voter0 [index $voters_all [expr $i*3]]
  set voter1 [index $voters_all [expr $i*3 + 1]]
  set voter2 [index $voters_all [expr $i*3 + 2]]

  set output0_net [get_nets_of_objects [get_pins [get_cells $voter0/0]]]
  set output1_net [get_nets_of_objects [get_pins [get_cells $voter1/0]]]
  set output2_net [get_nets_of_objects [get_pins [get_cells $voter2/0]]]

  disconnect_net objects $output0_pins
  disconnect_net objects $output1_pins
  disconnect_net objects $output2_pins

  connect_net net $output0_net objects $output0_pins
  connect_net net $output1_net objects $output1_pins
  connect_net net $output2_net objects $output2_pins
}
```

B.4 Early Split Constraints

An example TCL script used in Vivado to modify the placement of specific I/O cells within an FPGA design.

```tcl
# Create blocks
create_block pblock_fmc_data_a_0
create_block pblock_fmc_data_a_1
create_block pblock_fmc_data_a_2
create_block pblock_fmc_data_a_3
create_block pblock_fmc_data_a_4
create_block pblock_fmc_data_a_5
create_block pblock_fmc_data_a_6
create_block pblock_fmc_data_a_7
create_block pblock_fmc_data_a_8
create_block pblock_fmc_data_a_9
create_block pblock_fmc_data_a_10
create_block pblock_fmc_data_a_11
create_block pblock_fmc_data_a_12
create_block pblock_fmc_data_a_13
create_block pblock_fmc_data_a_14
```
B.5 Striping Constraints

An example TCL script used in Vivado to modify the placement of TMR domain-specific cells within an FPGA design.

```tcl
create_block block_dut_tmr; create_cells
resize_block block_dut_tmr
```
APPENDIX C. FPGA MEZZANINE CARD SCHEMATICS

This appendix contains additional information on the FMC coupler card as described in section 4.2.2. This appendix includes schematics that provide further details into how the I/O of the golden and test FPGAs are coupled together. The FMC coupler card is an essential part of the TURTLE platform.
Lanes are paired as follows:
[33:19] <-> [02:16]
[01] <-> [18]

Lanes 0, 17 are used for clocking and control signals