A Tool For Run Time Soft Error Fault Injection Into FPGA Circuits

A TOOL FOR RUN TIME SOFT ERROR FAULT INJECTION INTO FPGA CIRCUITS

BY

MARVIN ZUZARTE, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2014) McMaster University

(Computing and Software) Hamilton, Ontario, Canada

TITLE:	A Tool For Run Time Soft Error Fault Injection Into FPGA Circuits
AUTHOR:	Marvin Zuzarte B.Eng., (Mechatronics Engineering) McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Mark Lawford and Dr. Nicola Nicolici
NUMBER OF PAGES:	xiii, 90

I must dedicate this thesis to those that helped see me through it, as no worthwhile achievement can be accomplished without the assistance of others.

First to my parents, for supporting me through my life and education, always encouraging me to do whatever I am interested in, and sometimes misunderstanding but nodding and smiling all the same.

To my supervisors, for giving me the opportunity to work with them on this project, and providing me with guidance when I needed it.

To my sister and brother, for not making too much fun of me, and at least knowing I do things with computers.

Finally to my friends, for understanding whenever I had to tell them I couldn't do something because I was busy with research.

Abstract

Safety and mission critical systems are currently deployed in many different fields where there is a greater presence of high energy particles (e.g. aerospace). The use of field programmable gate arrays (FPGAs) within safety critical systems is becoming more prevalent due to the design and cost benefits their use provides. The effects of externally caused faults on these safety critical systems cannot be neglected. In particular, high energy particle striking a circuit can cause a voltage change in the circuit known as a soft error. The effects these soft errors will have on the circuit needs to be understood in order to ensure that the systems will function properly in the event soft errors do occur.

In this thesis a tool is designed to facilitate the run-time injection of soft errors into a hardware circuit running on a FPGA. The tool allows for the control over the number of injections that can be performed and control over the rate that the injections will occur at. Additionally the tool records time stamps of when injections occur and time stamps of when errors are detected. This recorded data allows for the analysis of designs in conditions prone to soft errors. The implemented tool allows for design time parametrization and run time configuration, allowing a multitude of tests to be run for a single compiled design. The tool also eliminates the need for a host computer after configuration by generating the injection locations and times on

the FPGA. Eliminating the host computer allows for faster testing when compared to other methods as data transfer times are greatly reduced.

The implemented tool was run on classical examples of redundant structures, such as duplication with comparison and triple modular redundancy as well as a non-redundant structure to establish a baseline. The results of multiple tests run on each structure are analyzed to illustrate the uses of the tool and how the tool may be used to test other designs.

Notation and Abbreviations

Abbreviations

ASIC:	Application Specific Integrated Circuit	2
COTS:	Commercial Off The Shelf	4
DWC:	Duplication With Comparison	18
FIT:	Falure In Time	5
FSM:	Finite State Machine	43
HDL:	Hardware Description Language	2
ID:	Identification	35
IOE:	Input/Output Element	9
LAB:	Logic Array Block	8
LE:	Logic Elements	1
LFSR:	Linear Feedback Shift Register	20
LUT:	Lookup Table	1
MCU:	Multiple Cell Upset	15
MLV:	Minimum Length Value	37
MTTF:	Mean Time To Failure	5

MUX:	Multiplexer	11
PLL:	Phase Lock Loop	9
rad-hard:	Radiation Hardened	4
RTL:	Register Transfer Level	17
SEFI:	Single Event Functional Interrupt	15
SET:	Single Event Transient	3
SEU:	Single Event Upset	3
SRAM:	Static Random Access Memory	2
TMR:	Triple Modular Redundancy	18
VLSI:	Very-Large-Scale Integration	22
XOR:	Exclusive Or	20

Contents

List of Figures

Chapter 1

Introduction and Problem Statement

This chapter will introduce the topics associated with this thesis, providing a basic understanding of concepts fundamental to understanding the work. This includes an introduction to field programmable gate arrays, single event effects, verification necessary for safety critical systems and the difference between faults, errors and failures.

1.1 Field Programmable Gate Arrays

Field programmable gate arrays (FPGAs) are reconfigurable hardware devices that allow for the implementation of custom digital hardware designs. These designs are primarily implemented using what the manufacturers of the FPGAs refer to as *logic elements (LEs)*; these LEs are comprised of *lookup tables (LUTs)* and register. The LUTs function as programmable truth tables allowing for the design of any digital

logic circuit within the constraints of the particular device. The design process for a FPGA is similar to that of an *application specific integrated circuit (ASIC)*, a design is written in a *hardware description language (HDL)* and a netlist is derived from the HDL circuit. The netlist is then mapped to the routing logic and LEs of the select FPGA creating a device specific binary configuration file. Programing of the FPGA is achieved by writing the configuration file to reconfigurable *static random access memory (SRAM)* within the device. The SRAM cells implement the truth tables of the digital logic circuit and also control the routing of the signals through the interconnect fabric within the device, allowing for the connection of the LUTs as defined in the netlist and required for the proper circuit operation.

For the rest of the thesis the terms register, flip-flop and flop will be used interchangeably.

1.2 Single Event Effects

A single event effect is a model of the effect of a high energy particle striking a device. When and where the particle strikes the device determines what effect, if any, it will have on the circuit (Battezzati *et al.*, 2010). The effects can be broadly separated into two categories soft errors and hard errors.

When a particle strike has enough energy to physically damage the device it strikes, it is known as a hard error. Hard errors change how the circuit operates, usually creating a permanent short between power and ground, leading to a constant current draw within the device. This type of error is permanent, irreparable and may lead to total device failure. Hard errors will not be referred to further as there effects are not relevant within the scope of this theses.

A soft error is a non-permanent change of state or data. A soft error does not irreparably change the physical circuit, damage is only done to the data not to the hardware. Soft errors can also be further divided into, among others, *single event transients (SETs)* and *single event upsets (SEUs)*, these will be described in further detail in section *2.2 Soft Errors*.

1.3 Mission and Safety Critical Systems

A mission critical system is a system in which an error can potentially lead to a catastrophic failure, causing substantial financial, property or environmental damage. One specific type of mission critical system is a safety critical system, the failure of a safety critical system can directly lead to the loss of human life, as well as the previously stated damages.

Modern technology contains many applications of safety critical systems ranging from relatively simple designs like seat belts to highly complex designs like fly-by-wire systems and, nuclear shutdown systems (Knight, 2002). The more complex systems have traditionally relied on software running on microprocessors or microcontrollers for their implementation.

Due to the high reliability required by safety critical systems extensive verification must be done to ensure all the requirements of the design are met. Not only must the verification be performed on the implemented software but also on related sub-systems included but not limited to hardware used, operating systems, schedulers, libraries and compilers (Lawford *et al.*, 2010).

1.4 COTS Devices in Safety Critical

For a variety of reasons the adoption of *commercial off the shelf (COTS)* components in safety critical applications has become more prevalent. One reason for the adoption of COTS components is cost requirements inhibit the use of *radiation hardened (rad-hard)* and other robust designs (Civera *et al.*, 2002). Another reason for the adoption of COTS devices is the maturity of the tool chains used for development. Due to the specialized nature of rad-hard devices the commercial use and adoption rates are low, this leads to the tool chains not being as mature as the tool chains for COTS devices. This immaturity of rad-hard tool chains can introduce problems in development including longer product cycles and the inability or difficulty to perform proper verification that is essential to safety critical systems. The issues that would be introduced in the development process because of the tool chain immaturity can potentially exceed the benefits that a rad-hard device would alleviate.

1.5 Verification of Safety Critical Systems

If a safety critical system fails there is intimidate dangers to human life, the potential loss or irreparable damage to a system. This high cost if a failure occurs leads to the need to ensure that any safety critical systems always function properly and there are no undefined outcomes. To ensure a failure does not occur safety critical systems undergo rigorous verification to ensure there proper operation. This verification process begins at the concept phase, includes all aspects of the system and continues till the final implementation.

One part of this verification process is how susceptible the system is to soft errors.

Manufactures use the term *falure in time (FIT)* to describe a devices susceptibility to soft errors. FIT is defined to be the number of failures per billion hours of operation, typically the FIT values range from 1 to 100. These FIT values equate to a *mean time to failure (MTTF)* of approximately 1140 to 114000 years (Spilla *et al.*, 2011). Due to these long times between failures accelerated testing techniques are required to properly determine effects of soft errors.

1.6 FPGAs in Safety Critical Systems

The adoption of new technology into existing and entrenched fields is usually met with some resistance until the benefits of adopting the new technology outweighs the costs and risk involved with the adoption. In some cases the adoption becomes inevitable due to the inability to source end of life products for new developments or as replacements for aging components. The benefits of using FPGAs have begun to outweigh the related costs and adoption into safety critical systems has increased as a result. The benefits of FPGAs include; portability, true concurrency, simplified timing analysis, simplified verification and, simplified error detection (Lawford *et al.*, 2010). These benefits apply to any system design in general but safety critical systems require rigorous verification to ensure proper operation so any simplification of that process benefits designers and regulators. The details of the benefits of FPGAs are described in section 2.3 Benefits of FPGAs in Safety Critical Systems.

1.7 Faults, Errors and Failures

Although the words fault, error and failure are often used interchangeably the literature surrounding fault testing defines these three terms very specifically, a brief description of each term follows. A failure occurs when a system does not perform the task it was designed to perform. An error is defined as when an output from a component in a system deviates from the proper result. An error within a system can potentially lead to a failure. Finally a fault is an anomaly within a system caused by a source outside the system. A fault may produce an error and become effective, or not produce an error and be inactive (Carreira *et al.*, 1999).

For this thesis the terms fault, error and failure will be used as described above. Additionally, the term injection will be used interchangeably with the term fault to refer to an injected fault.

1.8 Problem Statement

Many safety critical systems are deployed in harsh environments including nuclear power plants and aerospace where the effects of SETs are non-negligible because of the greater presence of high energy particles. The test and verification process for these systems does not have many hardware based tools for the testing of SET caused faults on the data-path of a FPGA circuit. Section *2.6 Related Work* overviews methods that have been developed. This thesis develops a method to inject faults within a design to allow for accelerated fault testing.

The method developed attempts to improve upon current designs by reducing the time needed to perform tests. This run time reduction is achieved by reducing

the communication overhead involved with currently proposed testing methods. The reduction is achieved by preforming the selection of injection locations and times on the same FPGA the design is being tested on. The developed method also injects faults without any slowdown to the operating speed of the circuit. This full speed injection is important because often test regulators require testing to be performed at the intended operating speed of the circuit.

Chapter 2

Background

This chapter provides more detail on the topics covered in chapter 1, expanding upon them and providing more information. This chapter will also provide some background on methods used in the implementation, additionally providing a review of related work that has been done in the field of fault injection.

2.1 Field Programmable Gate Arrays

To understand the failure modes of field programmable gate arrays (FPGAs) a basic understanding of FPGAs must be established. As an example the Altera Cyclone III FPGAs and the corresponding Altera Quartus II tool flow will be looked at, as they are representative of the general FPGA architecture and tool flow.

2.1.1 Functional Architecture

Cyclone III devices are functionally arranged in a two-dimensional row and column architecture, illustrated in figure 2.1. The *logic array blocks (LABs)* each contain 16

logical elements (LEs), the LEs allow for the implementation of user logic functions. The interconnect fabric consists of a row, column and local routing structure utilizing programmable interconnect switches to route signals. The interconnects provide access to *phase lock loops (PLLs)*, M9K RAM blocks, embedded multipliers and *input/output elements (IOEs)*. The M9K blocks are user accessible embedded dual port SRAM memories with 9K bits of memory each, although total available memory is dependent on the specific device. The IOEs provide access to the device pins and support of multiple I/O standards. There is also a global clock network of up to 16 clock lines connected to up to 4 PLLs that run through the entire device and can provide clocks to all the resources in the device.

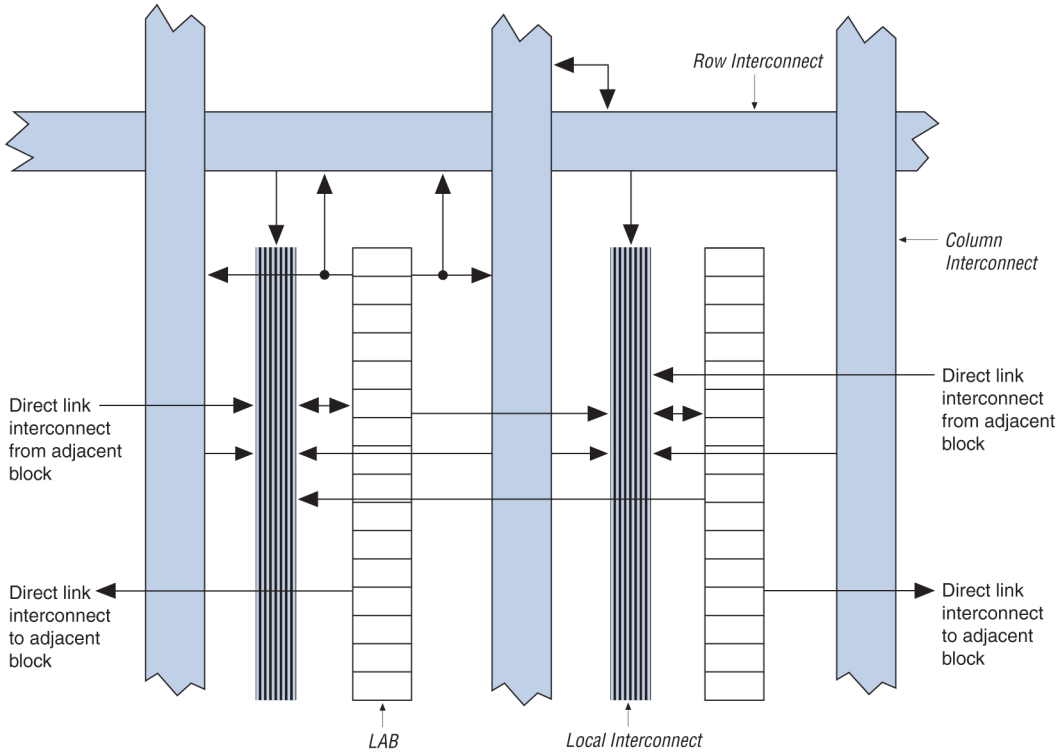


Figure 2.1: Cyclone III Device Block Diagram from (Altera Corporation, 2012)

2.1.2 Logic Elements

The basic building in FPGAs are the logical elements. Figure 2.2 is the circuit diagram for a Cyclone III LE. Within each LE there is a programmable *lookup table (LUT)*, a bypassable flip-flop as well as connections to the interconnect fabric. In addition to these features there is specialized hardware determined to be beneficial to commonly used designs. The LUTs allow for the implementation of arbitrary digital logic functions. How the design of the logical functions are achieved will be explained in section 2.1.3 Lookup Tables. The bypassable flip-flip allows for the implementation of combinational logic or sequential logic. For more information on the Cyclone III LE architecture see Altera Corporation (2012).

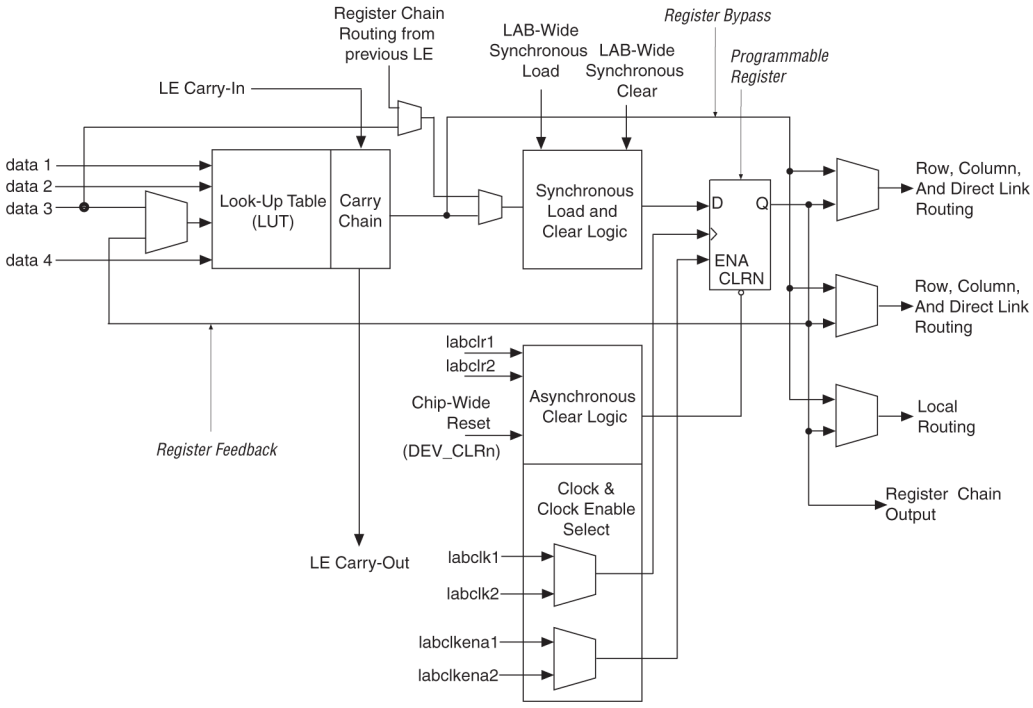
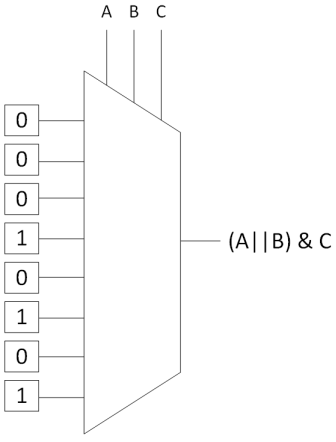


Figure 2.2: Cyclone III Logical Element from (Altera Corporation, 2012)

2.1.3 Lookup Tables

Lookup tables enable the FPGA to implement any arbitrary logic function a designer chooses; they are the fundamental building block of the FPGA logic. An N input LUT allows for the implementation of any combinational logic function of N or fewer inputs. Cyclone III FPGAs contain four input LUT, but it is common for higher-end devices to have LUTs with a greater number of inputs (Altera Corporation, 2007). Functions of greater than N inputs can be achieved by connecting multiple N input LUTs together, using the output of the previous LUT as input to the next. The way LUTs are implemented on FPGAs are by programming the result of the truth table into configurable SRAM cells connected to the inputs of a *multiplexer (MUX)*, the select lines of the MUX are then connected to the input variables of the logic function and select the output row of the truth table that corresponds to the input value. A 3 input LUT and a truth table for a function implementing $f(A, B, C) = (A \| B) \& C$ is shown in figure 2.3.



ABC	(A||B) & C
000	0
001	0
010	0
011	1
100	0
101	1
110	0
111	1

Figure 2.3: 3 Input Lookup Table

2.1.4 Tool Flow

The typical tool flow for a FPGA design is illustrated in figure 2.4, it begins with a *hardware description language (HDL)* design that is eventually mapped to the resources within the FPGA producing the implemented circuit. During the synthesis process the HDL is converted into a set of logic functions and interconnects called a netlist. The netlist then undergoes a technology mapping process where the netlist is mapped to the specific resources of the FPGA that the design is to be implemented on. This process creates the configuration file that specifies the values for all the configurable memory on the FPGA including the LUTs and the programmable interconnect switches. During the configuration process the FPGA is programmed with the configuration file which is also often referred to as the "bit-stream".

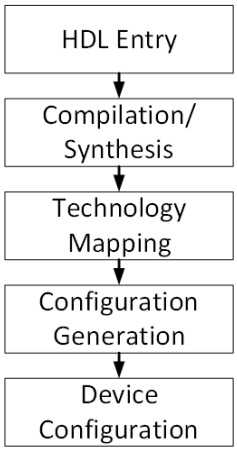


Figure 2.4: FPGA Tool Flow

2.2 Soft Errors

Now that a basic understanding of the design and mechanics of FPGAs has been established, how faults are induced in a device has to be investigated. Specifically in this case the investigating the occurrence of soft errors.

Soft errors are non-permanent errors induced in circuits by some sort of external interference. This interference can be caused by high-energy particles striking the circuit. Soft errors caused by particle strikes are becoming more probable because in an effort to decrease energy consumption and increase efficiency and speed, devices are using ever decreasing magnitudes of charge to carry and store information. As a result, this is leading to increasing probabilities that neutron and other high energy particle strikes can cause a transient error (Velazco *et al.*, 2007), (Civera *et al.*, 2002).

This section outlines some models of how soft errors occur and how they may affect a circuit. Most of the information is from *Reconfigurable Field Programmable Gate Arrays for Mission-Critical Applications* (Battezzati *et al.*, 2010). It can be referenced for further information on radiation effects on FPGAs.

2.2.1 Single Event Transients

When a particle strike causes a voltage variation to travel through the circuit, a *single event transients (SET)* has occurred. The quantity of charge the particle has, the angle it strikes the device at, the materials encounters, and the electric fields present at the moment of impact, all effect how it will affect the circuit. When a SET does occur, its duration normally ranges from picoseconds to nanoseconds. A SET is defined as a double transition $\{0 \rightarrow 1 \rightarrow 0\}$ or $\{1 \rightarrow 0 \rightarrow 1\}$. The value it transitions through is the polarity of the SET, i.e., if the transition is $\{0 \rightarrow 1 \rightarrow 0\}$

it is a positive SET, otherwise it is a negative SET.

The generation of a SET in a sensitive area of the FPGA may lead to a fault. For instance if a SET is generated on a functional logic trace it can be sampled into a memory element, introducing a fault that will further propagate through the circuit. If the SET is in the configuration logic, it could cause a temporary change in the routing of the circuit, then the change would immediately be restored after the SET ended. Potentially the most dangerous effect would be a SET on a global line, like a clock or a reset line where the whole circuit would be affected.

2.2.2 Single Event Upsets

A *single event upset (SEU)*, also called a *bit-flip*, is when a SET changes the value of a memory element like a flip-flop or SRAM cell. If a SET with a great enough amplitude is generated within a memory element, it could force a change in the feedback loop of the element inverting the value stored – hence the term bit-flip. Figure 2.5 shows a simplified SRAM cell where a SET is generated at the drain of the write enable transistor. If the original state of the cell was 1 then the SET would force a change in the input of the upper element which would then propagate to its output. Due to the feedback nature of the SRAM cell, the lower half would then be modified too, bringing the cell to the incorrect 0 state.

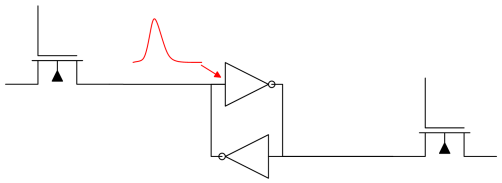


Figure 2.5: Simplified SRAM Cell Equivalent

SEUs are not normally permanent errors as the next time the cell is written to the

incorrect value will be overwritten. However in some situations where the memory element cannot be written to again, then the SEU becomes a permanent error until a global reset is performed or an error correction system corrects the error.

If a memory element affected by a SEU is read from or used in any other fashion (i.e. routing switch) before the value stored there is corrected this will cause faults to propagate through the circuit.

It is possible for multiple SEUs to be present in a circuit at one time through a process known as SEU accumulation. SEU accumulation occurs when a SET provokes a SEU and before the first upset is corrected, a second SET induces another SEU within the circuit.

2.2.3 Multiple Cell Upset

Another possible way multiple SEUs can effect the same circuit is when a single SET provokes multiple SEUs. This is known as a *multiple cell upset (MCU)*. If the incident particle has enough energy to travel a great distance through the device it can potentially travel through multiple memory cells and cause a MCU. The distance between memory elements is a fundamental parameter in the probability of an MCU occurring. The industry trend towards reducing feature size and higher device integration by decreasing the spacing between cells is increasing the probability of a MCU occurring, making MCUs a real concern for safety critical applications.

2.2.4 Single Event Functional Interrupt

A *single event functional interrupt (SEFI)* occurs when a SEU effects a memory element that is involved in the control of other processes, such as program counters

or a finite state machine state register. An upset in a location like these may lead to the device entering a state it cannot recover from unless a global reset is applied.

2.3 Benefits of FPGAs in Safety Critical Systems

There are many benefits to using FPGAs in safety critical applications as their implementation in place of software based systems simplifies many aspects that need to be verified in a safety critical systems. The implementation of safety critical systems utilizing FPGAs also eliminate some potential failure conditions that only occur in software implementations, such as deadlocks. This section goes over some of the benefits discussed in *McSCert Feasibility study of FPGA Based Platforms for Safety-Critical Systems* (Lawford *et al.*, 2010).

2.3.1 Portability

As devices age they need to be replaced, if identical parts cannot be sourced a replacement part must be found. This replacement part then must undergo verification testing to ensure the new part will be reliable. Often modifications to the related software are needed accompanying a hardware change. These modification to the software will require the software be re-certified too, incurring additional costs. Because HDL code is device independent and tends to be very modular in design it is easily ported from one device to another without modification. This simplifies the verification process greatly because the functionality of the circuit has already been verified. Additionally embedded components that would require testing if a new device was adopted such as a UART could be implemented as their own modules

eliminating the need to verify functionality of embedded components for every new device.

2.3.2 True Concurrency and Simplified Timing Analysis

Safety critical systems are usually required to perform multiple independent tasks repetitively, as is common for many embedded systems. In a typical software implementation operating on a microcontroller or general purpose processor only a single task can be executed at a single time. To meet the timing requirements of all tasks some sort of context switching scheduler must be used. It becomes increasingly difficult to ensure all deadlines are met as tasks are modified and added to the design. Additionally if there are any asynchronously occurring tasks in the design these can further complicate timing analysis especially if the design utilizes cached memory. This complexity makes pre-runtime scheduling very difficult to perform and even more difficult to correctly formally verify.

The benefits of FPGAs in the context of concurrent task scheduling is each task is synthesized into its own circuit. Each task being its own circuit allows concurrent operation, no context switching is necessary. This greatly simplifies the tasks of scheduling as it can independently be determined if each task will meet its deadline. There is also no longer a need to verify the function of the scheduler because a scheduler is no longer required. Another advantage is because HDL code is easily translated to *register transfer level (RTL)* descriptions longest paths can easily be determined, there are no paths that may be obscured from a tester such as branches or loops that may not get exercised during testing. FPGA vendors have accurate timing models of propagation delays in the devices so worst case timing can easily

be determined. These simplifications make verification easier and the overall system more reliable.

2.3.3 Simplified Error Detection

In order to mitigate against errors in any design key components or variables in the design must be replicated and compared to determine if a discrepancy has occurred. In a software implementation this may be achieved through repetition, where the same operation would be performed multiple times and the results compared. This repetition increases run time, incurring a time penalty. For a FPGA based design this replication of results can be done using additional hardware resources rather than repetition. This use of hardware resources rather then a longer execution time is desirable because in most time constrained systems hardware resources are less valuable then time.

Two such methods for spatial redundancy are *duplication with comparison (DWC)* and *Triple modular redundancy (TMR)* and are discussed further below.

Duplication With Comparison

DWC is a fault detection technique that uses redundancy to detect errors. In its simplest form an exact replica of the circuit is created and the outputs of both copies are compared to determine if an error has occurred. A more complex implementation of DWC could be created by comparing the values of all corresponding flops in the design to allow for earlier detection of a fault.

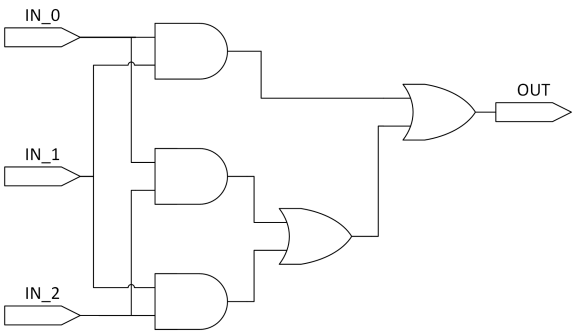
DWC does not provide any fault correction method as there is no way to determine in which copy of the circuit the fault occurred in. Thus DWC is only capable of

detecting if a fault has occurred but not capable of correcting it.

Triple Modular Redundancy

TMR is a fault detection and error correction technique that uses redundancy to perform the detection and correction. It's implementation creates three copies of the original circuit. The final outputs from the three copies are provided to a majority voter or majority gate. The output from the majority gate is the majority value of the inputs. Figure 2.6 shows the design and truth table of a majority gate. TMR designs ensure that the final output is correct so long as at least 2 of the 3 copies are providing the correct output.

A further extension to TMR is to perform majority voting on all memory elements in a design. This could potentially prevent a simple error from compounding and creating more errors such as in the case of a SEFI.



IN_0	IN_1	IN_2	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure 2.6: Majority Gate and Truth Table

2.4 Linear Feedback Shift Registers

Often pseudo-random number generation is utilized for some part of the test procedure. A typical and commonly used pseudo-random pattern generation structure is a *linear feedback shift register (LFSR)* (Ubar *et al.*, 2011).

Figure 2.7 depicts the structure of a simple LFSR. It consists of a series of flip-flops connected sequentially. The input to the flip-flop chain is the *exclusive or (XOR)* value of some of the registers in the chain. The registers that are chosen for the feedback are called taps. These taps determine what the next input value will be. The feedback terms can be expressed as a polynomial where each flip-flop in the LFSR has a term in the polynomial corresponding to its position. This polynomial is known as the characteristic polynomial of the LFSR. If the flip-flop is used in the feedback its constant is 1, otherwise it is 0. The final term is 1 and has no corresponding feedback or flip-flop. The characteristic polynomial for the LFSR in 2.7 is

$$P(x) = 1x^4 + 1x^3 + 0x^2 + 0x^1 + 1 = x^4 + x^3 + 1.$$

As the LFSR is clocked it will go through a sequence of unique states. The order of the states is determined by its starting or seed value and the taps selected. The maximum number of states an n bit long LFSR can have is $2^n - 1$. A LFSR is considered maximal if it goes through all $2^n - 1$ states starting from any seed value except the all zero seed. For further information on LFSRs a good reference is Automated Synthesis of Phase Shifters for Built-In Self-Test Applications (Rajski *et al.*, 2000).

For the rest of the thesis unless otherwise stated the terms random and pseudo-random will be used interchangeably to mean pseudo-random.

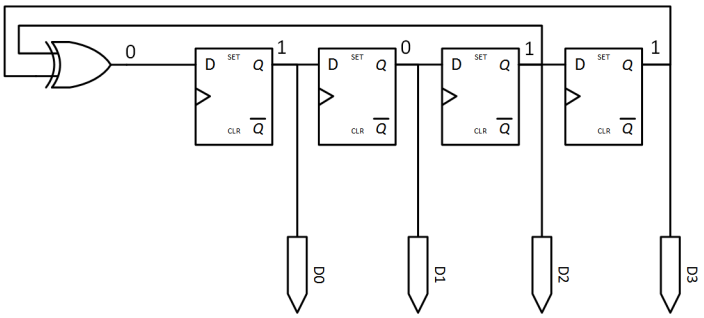
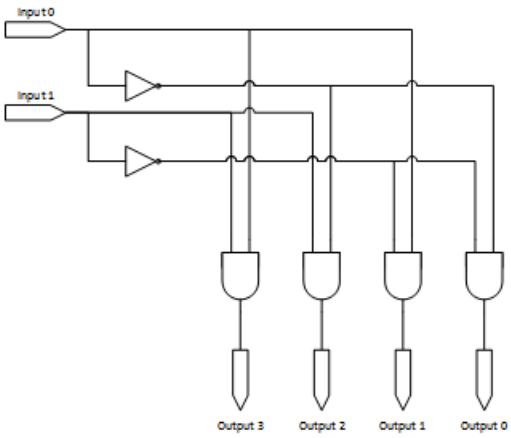


Figure 2.7: Basic LFSR

2.5 One-Hot Codes

A one-hot code is a binary vector whose only valid states contain only a single 1 in the vector, the rest of the values in the vector are 0. One-hot decoders take a n-bit input and produce a 2^n-bit output containing only a single 1 in the output. One-hot decoders are often used to select or enable a device on a shared communication bus, and are often called address decoder or line decoders. Figure 2.8 depicts a simple 2-bit input one-hot decoder and its input and output sequence.



Input	Output
00	0001
01	0010
10	0100
11	1000

Figure 2.8: Basic One-hot Decoder

2.6 Related Work

Most of the work done on fault injection into hardware circuits has been to investigate the robustness of *very-large-scale integration (VLSI)* circuits when used in to mission critical applications. The techniques used for SEU testing of VLSI circuits are still applicable to testing of FPGA based design with little or no modification. As FPGAs have become more powerful and accessible techniques utilizing FPGAs have been developed to accelerate the testing of VLSI designs similarly to approaches proposed in (Civera *et al.*, 2002) and (Sauer *et al.*, 2011), where an FPGA is used to emulate a modified MIPS32 processor to allow for fault injection. More recently as the benefits of FPGAs used in final implementations has become more prevalent more methods to test the robustness of the FPGAs themselves have been developed. These fault injection techniques can be generalized into three main categories; simulation based, physical and emulation based.

2.6.1 Simulation Based Fault Injection

Simulation based fault injection methods are similar to register transfer level simulation or other abstraction layer simulation programs. Simulation based methods can further be separated into two main categories; methods that require modification of the HDL code and methods that utilize commands built into the simulator.

Methods that modify the HDL code insert what are referred to as saboteurs into the circuit being tested at locations that faults would be injected into. During a fault injection the saboteur modifies a value or timing characteristic depending on what is being tested, and during normal circuit operation the saboteur is inactive. In the simplest form the saboteur is inserted between a driver and the corresponding

receiver. This placement enables the injection of faults in between the two points during the simulated operation of the circuit.

The approach that uses built-in simulator functions generally provides better performance than the method that relies on inserting saboteurs. This method then requires either a simulation tool that allows for fault injection or the access and ability to modify the simulators source code to enable this functionality. The ability to use this technique to perform fault injection relies heavily on commercial simulation tools and their ability to implement fault injection. This approach uses simulator commands that either manipulate signals within the simulated circuit or manipulate variables in behavioral simulation to inject faults.

Benefits of using simulation based fault injection are: it provides the maximum amount of controllability and observability and allows for testing at any abstraction level depending on the simulation method used.

The main drawbacks to simulation based fault injection methods include: The accuracy of the simulation tool's interpretation of the model and the length of time required to perform the tests. Depending on the type of simulation performed, the complexity of the circuit and the number of input vectors applied simulation methods can be 10^1 to 10^4 of times slower than physical or emulation based testing methods (Civera *et al.*, 2002). This time cost is compounded by the number of possible faults that can be tested and the number of tests that need to be run to ensure proper coverage.

Methods to speed up fault injection campaigns have been proposed. Berrojo *et al.* (2002) uses a technique that analyses the circuit to determine faults that will have equivalent effects after a given amount of time and determines faults in similar

structures. This is in order to collapse the number of faults that need to be tested.

2.6.2 Physical Fault Injection

Physical fault injection is a fault injection method that physically manipulates currents and voltages within the circuit to create faults. This is achieved in one of two ways, pin level manipulation and non-contact injection.

Pin level manipulation is regarded to be the first type of fault injection (Carreira *et al.*, 1999). Originally the technique used probes to simulate bridging and stuck-at faults. More modern examples use sockets inserted between the target hardware and its circuit board capable of simulating complex logical faults. The main limitation of pin manipulation is its limited accessibility to perform fault injections into the circuit. With increasing circuit complexity and package density this testing method must be augmented by some other method to provide adequate fault coverage.

Non-contact testing involves using a fault injection method that has no physical contact with the circuit being tested. Fault injection is achieved through exposing the circuit being tested to a source of ionizing radiation and observing the effects. This type of testing is often referred to as accelerated radiation testing (Fabula J., 2007). For this type of testing access to specialized, usually prohibitively expensive, hardware is required.

Because of SRAMs sensitivity to SEU induced faults this method of testing has been used to investigate faults in SRAM based FPGAs. In 2005 Xilinx revealed an ongoing program called the Rosetta experiment to measure real world background radiation effects on FPGAs (Lesea *et al.*, 2005). Xilinx has placed thousands of devices throughout 10 locations around the word and are monitoring them to observe

the effects of background radiation. Specifically Xilinx is looking for the effects on configuration memory and SEFIs. The Rosetta experiment is an experiment to quantify the effects of radiation on FPGAs rather than a specific method to test a design against the effects of SEUs.

Advantages to accelerated radiation include: experiments can be run in real time, it is possible to inject errors into otherwise inaccessible areas and no modifications need to be made to the target system.

Disadvantages to accelerated radiation testing include; the need for expensive, special purpose testing hardware to perform the tests, and there is almost no controllability or observability during the tests. Faults can be injected but where, when or how they have occurred cannot be controlled or accurately monitored.

2.6.3 Emulation Based Fault Injection

Emulation based fault injection methods implement a modified design on a FPGA to evaluate the effects of faults. Emulation based fault injection can be separated into two different categories, configuration memory modification and source modification.

Configuration memory modification, often called bitstream modification, performs synthesis on an unmodified design and generates a configuration bit stream. An external program is then used to modify the bit stream by flipping a bit in the bit stream. The modified bit stream emulates the effects of a SEU on the FPGA (Asadi *et al.*, 2003). This modified bit stream is then programmed to the FPGA, a test if then performed and the results analyzed. The need for a host computer to reconfigure the FPGA introduces overhead, the majority of time during the test is spent reconfiguration the FPGA. Efforts have been made to reduce this bottleneck and speed up the

testing process, Lima *et al.* (2001) and Cieslewski *et al.* (2010) propose and develop methods to perform multi-bit testing, targeting bit flips into unique complex logic blocks to speed up testing and avoid faults masking one another. Another method for speeding up fault injection campaigns is the use of partial reconfiguration (Nazar and Carro, 2012). Partial reconfiguration takes advantage of some FPGAs ability to reconfigure a relatively small portion of the entire device, this portion is referred to as a frame. The technique used by Nazar and Carro (2012) also eliminates the need for a host computer for anything but saving test results. This elimination of the host computer is achieved by generating the frame reconfiguration bitstream on the device being tested and using an internal programming port to perform the partial reconfiguration.

The greatest limitation to this bitstream modification is it is limited to faults in the configuration memory of the FPGA, this makes testing of transient errors difficult if not impossible. Also it assumes that the fault is present for the entire duration of the test (Nazar and Carro, 2012) (Lima *et al.*, 2001), because reprogramming the configuration memory will clear any previous state data that was there, corrupting the previous state.

Emulation testing requiring source modification is conceptually similar to simulation source modification where a saboteur is introduced into the circuit to enable fault injection. The main difference in the case of the emulation saboteur is the saboteur and modified circuit must be synthesizable.

Sauer *et al.* (2011) and Spilla *et al.* (2011) implement methods to inject faults into the registers of a MIPS32 processor emulated on an FPGA. This is achieved by implementing a shadow register for each register in the design then shifting ones

into the shadow register where a fault is to be injected. This shadow register is then XORed with the original register to create the bit flipped value. When an "injector" signal is applied the erroneous value is output instead of the unmodified value. Figure 2.9 depicts the circuit architecture implemented in Sauer *et al.* (2011) and Spilla *et al.* (2011). This design still requires a host computer to control the shift chain for selecting fault locations and controlling the execution of the test. The bottleneck from the host computer is present but is minimized when compared to bit stream modification because only small amounts of data are transfers at a time. Although the maximum fault rate achievable by the design was not specified the maximum fault rate used in testing was 100 fault/s, a rate orders of magnitude higher than would be experienced in real world operation.

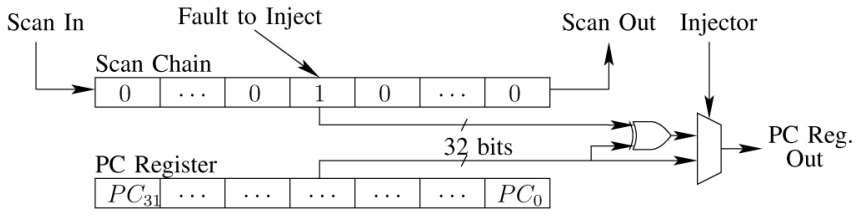


Figure 2.9: Shadow Register Fault Scan Chain Implementation from Sauer *et al.* (2011)

Civera *et al.* (2002) implements a method where every flop in the FPGA design is replaced with a modified flop design shown in figure 2.10. This modified flop design allows for the injection of faults into any flop in the design and the ability to record the state of every flop in the design at every clock cycle if desired. This design also requires a host computer to load the scan chain and control the tests, once again becoming the bottleneck in performance of the design.

The advantages to source modification emulation fault injection are; tests can be

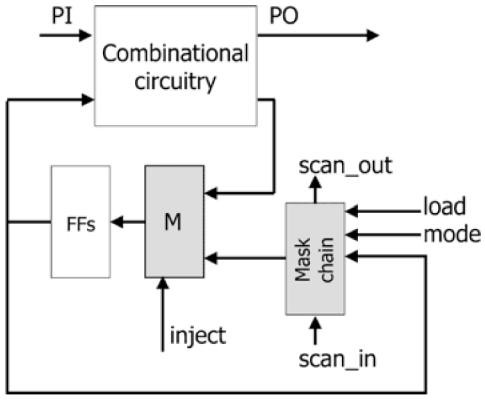


Figure 2.10: Instrumented Flop from Civera *et al.* (2002)

run at the operating speed of the circuit and there is no limitation to at what point during test execution a fault can be injected.

The main disadvantage to source modification emulation fault injection are: the complexity of the tool required to modify a HDL design while still allowing it to be synthesizable is high. Another disadvantage is only the functional steady state consequences of a fault can be analyzed, temporal effects are difficult to test for because faults can only be injected on clock edges.

Chapter 3

Implementation

3.1 High-level Overview

For a circuit design mapped to a FPGA the tests that can be performed should be able to encompass the effects of a SEU striking a user memory element. Ideally a tool that allowed for testing the effects of SEUs on FPGA circuits would be able to support several fault modes. The minimum of which is the ability to simulate the effects of a SEU affecting a random location in the design. From that ability would stem the functionality to simulate multiple random SEUs in order to simulate the effects of SEU accumulation. With the capability to simulate multiple SEUs the ability to control the maximum number of SEUs is then required. Targeting errors to user specified locations in the design would then be another useful feature to allow for testing potential critical points within the circuit. Then from the previous methods the combining of random selection while simultaneously targeting specific locations can also be used to investigate weaknesses. With the ability to control the number and location of simulated SEUs another important parameter to control becomes the

rate at which SEUs are simulated at, resulting in the need to be able to control the timing of injections. The tool must also have a way of recording relevant data because being able to simulate SEUs is of no consequence without having results to see what is being affected.

A tool that supports all the above features and will not slow down the operational speed of the circuit being tested would have to be implemented on the same FPGA as the circuit being tested. In order for a fault injection tool to simulate SEUs in elements within the circuit it must be able to directly modify the original circuit to insert structures to enable fault injection. The control hardware that drives the injection structures at the correct times would then be wrapped around the modified design as shown in figure 3.1. Due to the fault injection tool operating directly on the FPGA a host computer would then be required to transfer test parameters to the fault injection tool. These parameters would allow for the control over the various functions of the tool including control of the number of faults to inject, rate at which faults are injected and, if specific locations are to be targeted *etc.*

With the control hardware wrapping the modified circuit the host would send instructions to the injection controller to set up the conditions for the test to be run. The control hardware would then signal the injection sites to inject at the appropriate times. Input stimuli to the circuit would not be modified in any way to ensure any recorded error is from the simulated SEUs not from other sources. The output of the circuit is monitored but not modified to allow for the recording of trace data.

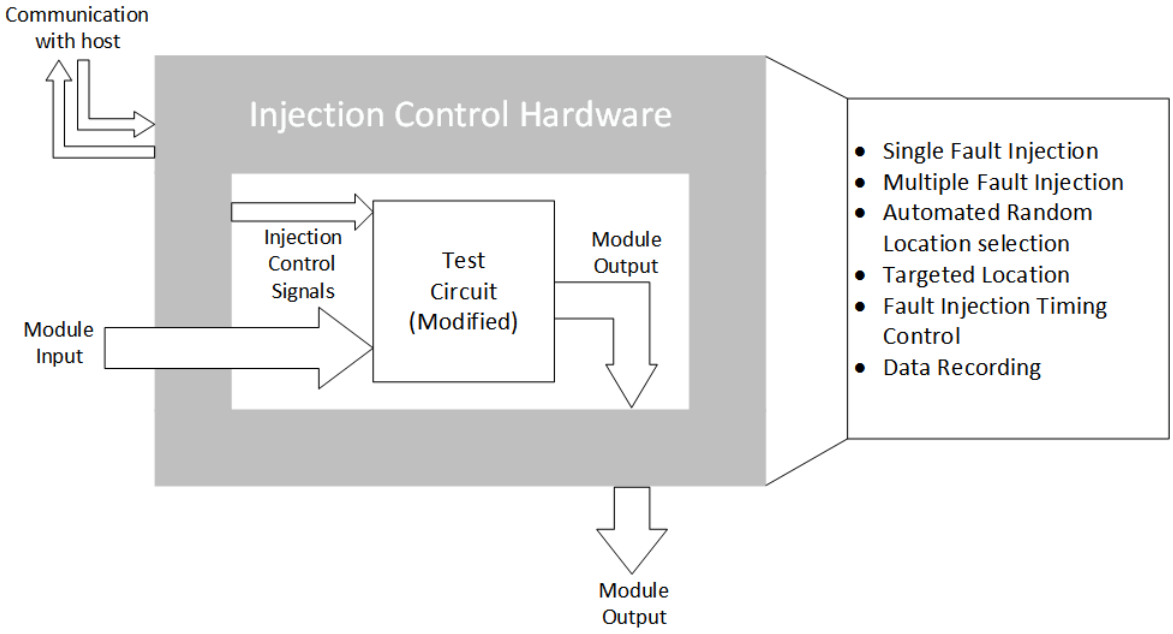


Figure 3.1: High Level Tool Diagram

3.2 Objective

The objective of the developed tool is to enable fault injection into circuits running on an FPGA. The purpose is to simulate the effects of a SEU, allowing for the investigation of the effects of SEUs on a circuit. The developed tool modifies an existing design to enable fault injection into that design, and supports the injection of multiple faults, and the control over the interval between when the faults are injected. When the effect of an error is detected the tool records the time it was detected along with other data described in section *3.12 SRAM Data Collection* for further analysis. Another important feature is the ability to configure at runtime the number of faults injected and the interval between injections, this allows for the modification of the test parameters without the need to regenerate and recompile the design. The tool can also be instructed to target specific flops in the design, thus

every injection cycle the targeted flops will have an error injected into them, and any remaining flops will be randomly selected.

3.3 High-level Operation

The operation of the implemented tool is as follows. A user will input a design to modify and specify the maximum number of injections to occur per interval and the initial period of that interval. Then the tool will modify the existing HDL code and generate all the other required HDL files to control injection, and generating a list of the flops that have been modified. After the generation the original design will have to be manually modified so that the modified module replaced the specific instance of the module that will be tested.

When enabled by a begin test signal the injection circuit will then operate injecting faults at the specified rate. When the memory on the FPGA is full or the process is manually stopped the test results can be exported for analysis, the test can then be reset or continued to acquire more data.

To perform the runtime modification a custom translator program is used. The translator generates a memory configuration file from a user configuration file. The configuration file specifies the number of injection sites, the period of injection, and allows the user to specify specific flops to target. This memory configuration file is then programmed to the FPGA, allowing tests to be run with the new configuration.

3.4 Tool Operation

The implemented tool modifies a Verilog design to allow for runtime fault injection simulating the effects of a SEU or MCU in the user flip-flops of the modified design. The tool does not modify the clock paths in the design being tested and does not modify the data input to the flops during the asynchronous reset state. The tool allows for control over the number of faults injected, and the frequency at which the faults are injected. This allows the user of the tool to specify how many faults and the rate that they want them to be injected into the module being tested.

The design also allows for runtime modification of the injection interval and the number of faults to be injected through a programmable memory. This memory also allows for the targeting of errors to specific flops in the design. When no user specified fault injection sites are designated or the number of sites specified is less than the number of total injection sites, then the rest of the injection locations are pseudo-randomly selected.

The tool operates similarly to what is described in section *3.1 High-level Overview*, creating a wrapper around the modified design containing all the control hardware. The tool also instantiates a copy of the original module that is provided the same inputs as the modified module. This original module provides error free outputs that are compared to the outputs from the modified module to determine if a injected fault has created an error. Figure 3.2 illustrates this comparison and error signal generation.

The remainder of this chapter will elaborate on the specific mechanisms involved in the operation of the tool and the rationale behind some of the design choices.

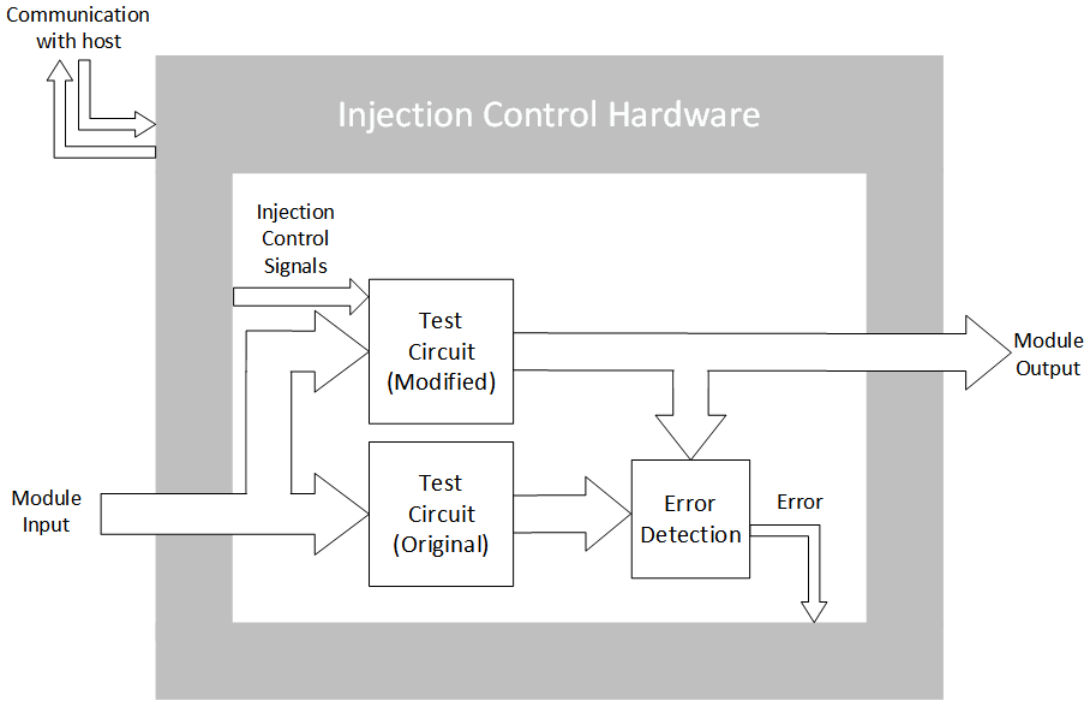


Figure 3.2: High Level Implementation Diagram

3.5 Source Modifier

The source modifier is what takes the original HDL design and modifies it to allow for faults to be injected into the flops. This is done by inserting a saboteur structure just before the flops' data input that allows for the value to be inverted.

The source modifier reads an input Verilog file and modifies the existing flops in the design, inserting the saboteurs that allows for a fault to be injected. It is assumed that before fault injection testing the module has already been verified to function correctly, therefore the module has already been compiled and there are no syntax errors, thus the source modifier does not verify that the module's design is syntactically correct.

The parser is not able to parse packed arrays so any packed arrays in a design need

to be converted into unpacked arrays before the tool is run. Additionally to exclude lines from the source modifier the comment "// source_mod_skip_line" can be added to the line. This comment must be added to register lines within the asynchronous reset state of an always block to instruct the source modifier to not modify those lines. The need to exclude the asynchronous reset states from the modification is because if the reset states are modified the compiler generates latches and gated reset signals. The latches and gated resets can potentially cause timing violations and modify the functionality of the original designs.

After the input file is passed to the source modifier it goes through the file and creates a list of all the registers in the design, and a corresponding list of register bit-widths. After the registers are found, the source modifier assigns an *identification (ID)* number to each of the flip-flop in the register, this ID is called the *flopID*. Only the flopID for the 0th bit in each register is recorded because the others flopIDs are determined from that ID using the bit positions of the flip-flop in the register. The flopIDs are not assigned linearly but rather as powers of 2, resulting in IDs of 2,4,8... *etc.* and the flopID of 1 is reserved to allow for no injection to occur. The total number of flip-flops found is reported to the software responsible for generating the rest of the control hardware. Numbering as powers of 2 allows for the use of one-hot codes when addressing the injection sites, enabling the use of a centralized address decoder rather than a decoder at each of the injection sites. Using a centralized address decoder reduces the amount of hardware required to perform multiple injections because one hot-codes can easily be combined to create a multi-hot code to rather than having n number of decoders at every flop to support n number of simultaneous injections. Multi-hot codes are discussed more in section 3.3 Addressing

the Flip-Flops.

The saboteur structure that replaces the original flop in the design in illustrated below in figure 3.3. The structure uses a XOR gate to simulate the effect of a SEU when the corresponding flopID signal is asserted. Normally the flopID signal is not asserted and the circuit operates normally, when the flopID signal is asserted the fault is injected by inverting the input value of the flop.

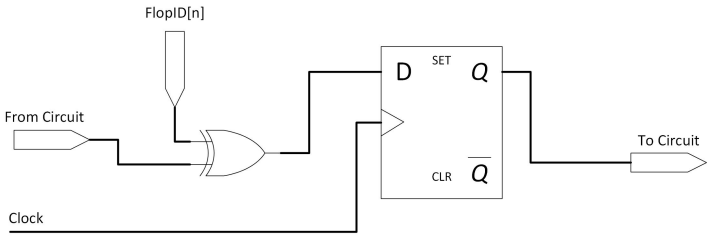


Figure 3.3: Instrumented Flip-Flop

3.6 Random Flop Selection

The pseudo-random selection of which flip-flop to inject the errors into is done using a LFSR and some ranging logic to ensure the value is within the correct range for the module being tested.

The implemented LFSR is 63 bits long and was chosen because it requires only 2 taps to be maximal, keeping the feedback logic to a minimum. The characteristic polynomial of the implemented LFSR is $P(x) = x^{63} + x^{62} + 1$. A 63 bit maximal LFSR also will only repeat after $(2^{63} - 1)$ clock cycles, at 50MHz this is approximately 5849 years (Alfke, 1996). This duration is more than long enough for the purposes of this fault injection tool. The LFSR design is illustrated in figure 3.4, it also allows for the loading of a specified seed, and incorporates a update signal so a new LFSR

value is only generated when necessary.

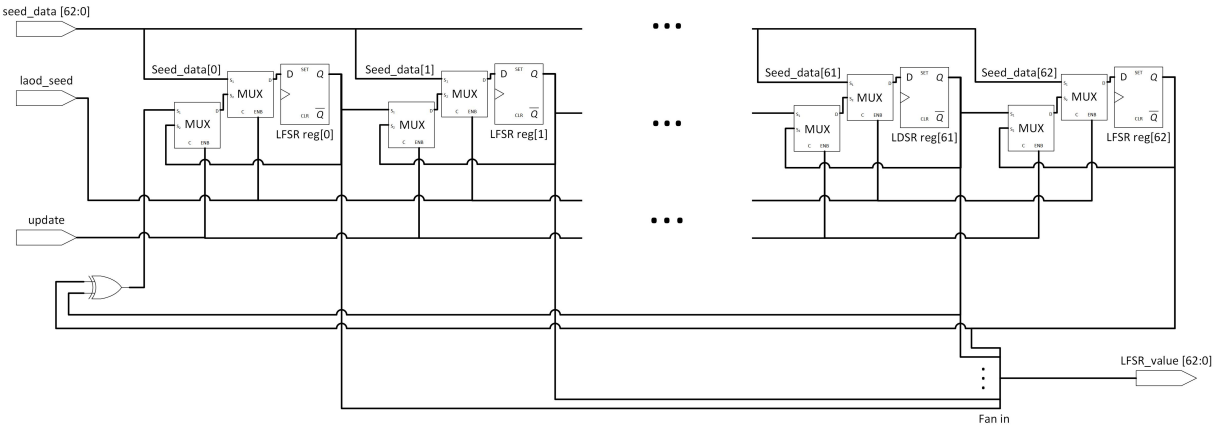


Figure 3.4: Implemented LFSR

The ranging logic is designed to be simple, require as few hardware resources as possible, and ensure the generated values are always less than or equal to the total number of flip-flops in the design. This is achieved by reading the minimum number of bits from the LFSR necessary to represent the number of flip-flops in the design i.e. reading only $\lceil \log_2 k \rceil$ bits to represent a design with k flops. We will call this read $\lceil \log_2 k \rceil$ bit value the *minimum length value (MLV)*. In the case that the number of flops in the design is not a power of 2 there are still $2^{\lceil \log_2 k \rceil} - k$ values that are invalid. To ensure these invalid values are not used the MLV is checked against the number of flops in the design and if the MLV is greater in the total number of flops the most significant bit is changed to 0, this makes the MLV less than the total number of flip-flops in the design and brings the value into the correct range, figure 3.5 illustrates the design of the of the ranging logic.

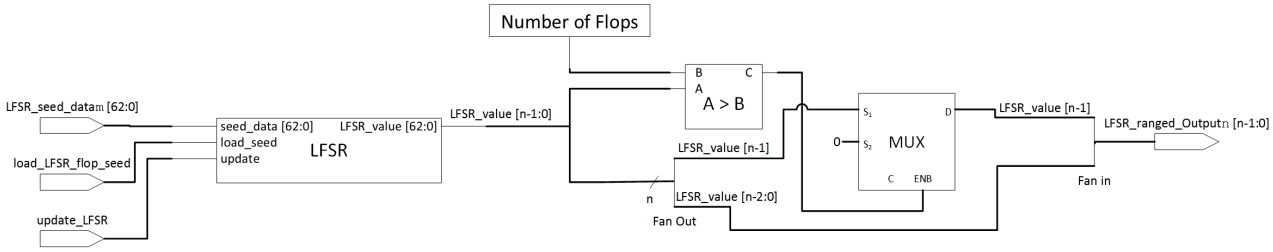


Figure 3.5: Ranging Logic

Using this naive method to keep the generated values in the correct range introduces a bias into the pseudo-random numbers generated. The reason the decision was made to range in this way was because if a statistically unbiased method was to be used it would either consume too many hardware resources to be practical or would not be guaranteed to generate a valid value within a single clock cycle.

3.7 Addressing the Flip-Flops

After the pseudo-random number is generated and ranged the value has to be mapped to the flopID number assigned during the source modification process. The way this is achieved is by using a one-hot address decoder resulting in the ability to simply address each saboteur individually.

Using one-hot decoders requires $2^{\lceil log_2 k \rceil}$ wires to address k injection sites vs. $\lceil log_2 k \rceil$ wires needed if individual address decoders were used the benefit is a reduction in the total number of gates required. If the ranged value was sent to each injection site it would only require $\lceil log_2 k \rceil$ wires, but there would have to be an address decoder at each injection site. Having individual address decoders at each injection site would require $k(2^{\lceil log_2 k \rceil} * (\lceil log_2 k \rceil - 1))$ gates, where k is the minimum number of bits required to represent the number of injection sites. Where as having

one centralized address decoder only requires $2^{\lceil log_2 k \rceil} * (\lceil log_2 k \rceil - 1)$ gates. The design of a k to 2^k one hot decoder is illustrated in figure 3.6.

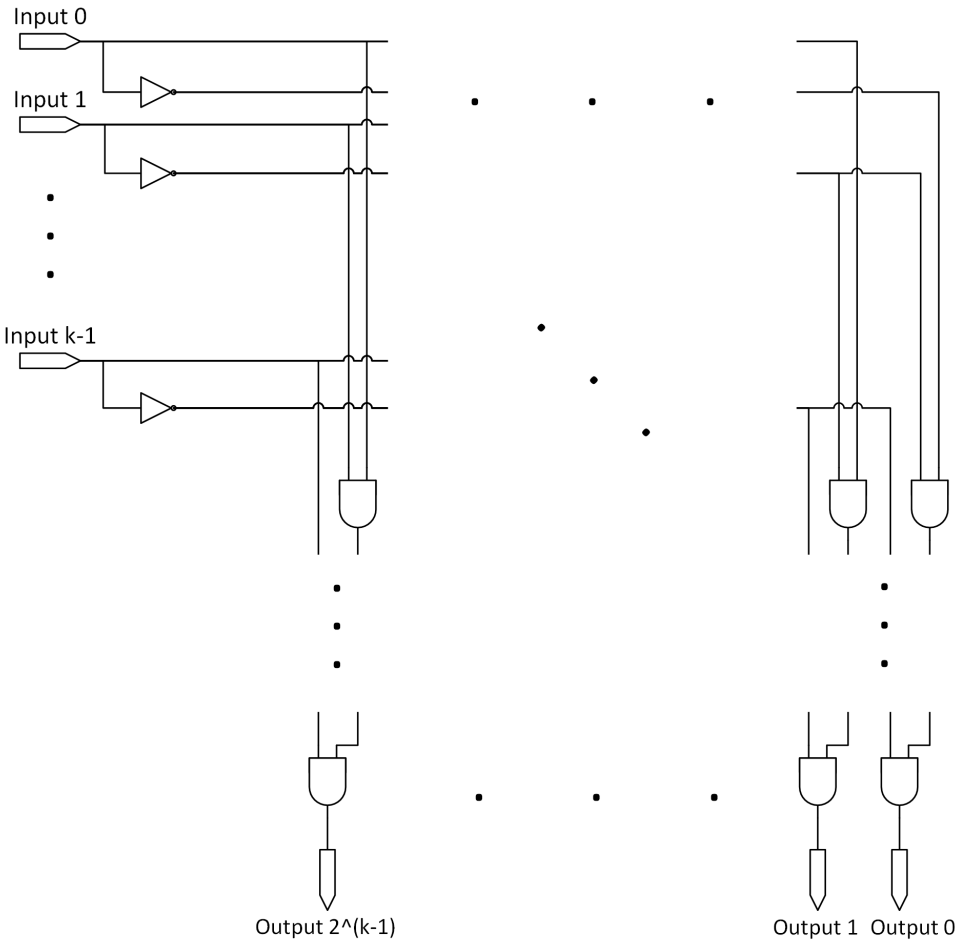


Figure 3.6: k Input One-hot Decoder

Using a common address decoder also has benefits when doing multiple simultaneous injections. If individual decoders were to be used for simultaneous injections the number of gates required would be $m * 2^{\lceil log_2 k \rceil} * (\lceil log_2 k \rceil - 1)$ opposed to $m * 2^{\lceil log_2 k \rceil} * (\lceil log_2 k \rceil - 1)$ for a common address decoder implementing the simultaneous injection decoding, where m is the number of simultaneous injections and k is

the number of injection sites.

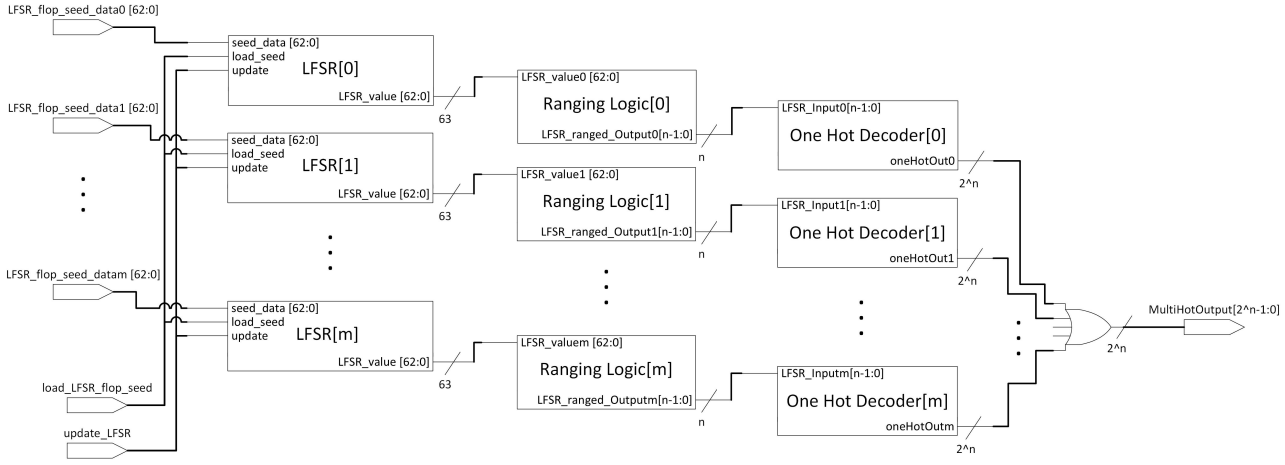


Figure 3.7: Multi-hot Decoder

3.8 Injection Timer

The injection timer is the module that controls when the errors are injected. This is achieved using programmable counters allowing for the control of injection periods.

For a fault injected into a circuit there is a chance that the fault can be masked. For example if the fault is injected one cycle before a new value is loaded the fault will likely be overwritten before it propagates into an error. This masking can become an issue because many hardware designs are rate limited designs. Rate limited designs load a value, preform an operation on the value, and produce a result. This process is repeated indefinitely with a fixed period. As a result of this static period if a fault is injected with a constant period between injections it is possible for the injected fault to be masked. This masking would occur if the fault is repeatedly injected at a time when it would be overwritten by new data. A fault periodically masked in this way

it would make the structure appear more robust then it may actually be. This leads to the need to inject faults at a random time but still at a predetermined rate.

To avoid the injection of faults at the exact same time, every cycle of the module under test the faults are injected at a pseudo-random time within the injection interval. For example, if a rate of 3 errors per 1023 clock cycles is specified, it would result in 3 errors being injected within 1023 clock cycles, not 3 errors injected exactly every 1023 cycles. This allows for simulating a fault rate that would be similar to what would be encountered in a real application, where the probability of a SEU would be expressed as a rate per unit time.

The way the pseudo-random time is determined is by generating a value from a LFSR. The injection timer uses a counter that counts from a programmable top value to 0 to time the injection period, and the LFSR value is used to perform the injection at a time within this interval. The problem exists again that the LFSR value can be greater then the maximum value of the injection period, so a way to insure that the LFSR value is less then this maximum is needed.

The way this maximum is ensured is by performing a bitwise AND between the LFSR value and the injection timer counter top value. Performing the ranging this way only operates properly if the top value is of the form $2^n - 1$, where n is an integer between 1 and 32. This form means the binary representation of the top value only contains 1s, when the AND operation is performed between the top value and the LFSR value the resulting value is guaranteed to be equal to or less then the top value. This raging operation limits the range of the injection intervals to numbers of the form $2^n - 1$.

3.9 Flow Module

The flow module is the wrapper that connects the other components together and contains the state machine that controls the loading of injection parameters and starting the injection campaign.

The program that generates the flow module is the primary script run, it calls all the other scripts that create the other hardware files and modifies the original source.

The flow module generator reads a configuration file called "inputconfig.txt" that contains:

- The location and name of the file that is to be modified
- The location and name of the Quartus II project file the module is in
- The max number of flops to inject faults into during one injection period that will be required
- The initial injection period
- The name of the error warning signal if one exists
- The name of any signals to be excluded from the error comparison

Using these values the flow module generator calls all the necessary scripts passing values necessary to generate the modules necessary for fault injection. First the source modifier is called passing the location and name of the file to modify, the source modifier modifies the input source, generates a list of all the modified flops for the user, and returns the number of modified flops to the flow module generator. From the number of modified flops the bit width required for the one-hot decoders are

determined. Using this bit width and the total number of injections per injection period specified, the multi-hot decoder is generated with a number of inputs equal to the total number of injections per injection period. The injection timer is then generated using the total number of injections to generate the appropriate number of LFSR comparisons. Next the LFSRs and the ranging logic for the random flop selection is created.

After all the ancillary hardware files are generated the wrapper module is generated beginning with the port declarations. Then all the registers for storing the programmable values are declared along with the wires necessary for routing all the signals. All the instantiations of the ancillary hardware are then made and the signals routed. The modified module and a copy of the original module are instantiated, and their outputs are compared to determine if an error has occurred. The SRAM memories are then instantiated along with a run-time counter. The run-time counter records the total time from the when the fault injection campaign was begun, this time is recorded to the memories at the various trigger conditions.

Finally the *finite state machine (FSM)* is generated to control the loading of all the programmable values, resetting of the timer values, LFSR seeds and starting the injection campaign. The next section elaborates on the operation and states of the FSM.

3.10 Injection Campaign Finite State Machine

The state machine that controls the fault injector has 4 states an idle state, a read SRAM state, a load signals state and a run injection state.

Idle

This is the default state and is entered on initialization or after a reset is applied. This state waits for a start signal to be applied in order to transition to the next state, Read SRAM.

Read SRAM

The Read SRAM state reads the values of the configuration data SRAM and loads the data into intermediate registers, this data is the programmable values that are described in section *3.11 Programmable Parameters*. For consistency this process takes 99 clock cycles to complete before moving to the load signals state.

Load Signals

The load signals state loads all the values stored in the intermediate registers to from the previous state in the the intermediate registers into there respective components. This process is done in a single clock cycle. This state also resets the the timers and memory address counters. When complete the state waits again for the start signal to be asserted to proceed to the Run injection state.

Run Injection

In the run injection state the timers and LFSRs function as designed and the injection campaign begins. The run injection state is never exited even after the memories are full, this is because if external data capture allowing for longer testing is used the test is not ended abruptly.

3.11 Programmable Parameters

There are certain parameters that can be specified during the generation process, and others that can be specified after the generation. This section will go over what the parameters are and their purposes. Some parameters cannot be changed after generation and compilation and others can be modified after compilation while the design is operating on the FPGA. These post-compile-time modifiable parameters allow a tester to perform multiple varied tests without the need to regenerate and recompile the design for each test.

3.11.1 During Generation

The parameters specified during the generation process are: the maximum number of of injections per injection cycle, the initial number of injections per injection cycle, the initial number of simultaneous injections, the initial injection period, if the module being modified has an error warning signal and the name of the warning signal, and the name of any signals to exclude from the error comparison.

Maximum Number of Injections per Injection Period (MIPI) This parameter determines the maximum number of saboteurs that can be addressed at a given time and is the upper limit on the number of injections that can occur in a single injection period. The range of valid values is [1,10].

Initial Number of Simultaneous Injections This determines the number of simultaneous injections that occur per injection cycle. The valid range for this value is [0,MIPI]

Initial Injection Period This determines the length of the period that the injection occur within. The valid values are [1,$2^{32}-1$], but the values must be of the form $2^n - 1$ as described in section *3.8 Injection Timer*.

Fault Warning Signal Name This indicates the name of the fault warning signal if one is present in the module. This signal name is required to create the write enable signal for the warning detection memory, and to ensure it is removed from the comparison for the error write enable memory, otherwise every warning generated would appear as an error even if an internal mechanism detected and corrected the fault.

Other Signals to Remove If other signals are to be excluded from the comparison that determines if an error has occurred they have to be specified.

Three things cannot be changed after the generation process, the first is the total number of injections per injection period, the second is the fault warning signal name and the third is the signals that are removed from the error comparison.

3.11.2 Post Compile modifiable Parameters

After generation the variables that can be configured are:

Number of Injectors to Use (NSIU) This allows for the modification of the test to allow for less than the maximum number of injections per injection period to be used. The valid range of values for this is [1, MIPI]

Number of Simultaneous Injections This specifies the number of simultaneous injections to occur. The range is [1, NSIU]. If this number is less than the

number of injectors to use the remaining injections will occur at random times within the injection period.

The Number of Specific Flops to Target This specifies the number of flops that will be specifically targeted for an injection. Range is [0, NSIU].

flopID Values For each flop specifically targeted flop a flopID value is required.

Injection Period This specifies the new injection period to use. This value is identical to *Initial Injection Period* in section *3.11.1 During Generation*.

Seed Values for the Injection Timers If a specific value is to be used for the injection timer LFSRs seed value it is specified here. The range is [1, $2^{64} - 1$] or [0] for a randomly generated value.

Seed Values for the Flop Selection If a specific value is to be used for the random flop selection LFSR seed value it is specified here. The range is [1, $2^{64} - 1$] or [0] for a randomly generated value.

These variable test parameters allow a tester to compile the design once, and then perform a number of varied tests by changing test parameters.

3.12 SRAM Data Collection

Depending on the configuration chosen during generation, two or three SRAMs will be instantiated for data recording. If a warning signal is present then all three SRAMs will be generated, one for recording whenever a warning is generated, one for recording whenever an injection occurs, and one for recording whenever an error is detected. If no warning signal is specified only the later two are generated.

Another SRAM is generated to allow for the storage and loading of the run-time configurable parameters. This memory is initialized with the "initial parameters during the first programming of the FPGA and can be loaded with different test parameters in order to perform alternate tests.

The uploading and offloading of data from the SRAMs is done using a tool within Quartus II called "In-system memory content editor". This memory management tool allows for the loading and reading of the SRAM on the FPGA without interrupting the operation of the device.

3.12.1 Determination and Recording of an Error Event

To determine if an injected fault has propagated to the output of the circuit the original unmodified module is instantiated operating in parallel to the modified module. The unmodified module operates unperturbed and will be referred to as the gold standard module. Both modules are provided the same inputs from the higher level module. The output from the gold standard is compared to the output from the module being tested to determine if an injected fault has propagated to the output of the module being tested. If a discrepancy is found between the gold standard output and the output of the module being tested, the time of the discrepancy is recorded as an error.

An error is defined as whenever the output from the modified module being tested does not exactly match the output from the original unmodified module. When this mismatch occurs the time since the start of the fault injection campaign is recorded in the error detection SRAM.

3.12.2 Determination and Recording of a Warning Event

Fault tolerant designs have internal structures that detect when an internal discrepancy has occurred and generate a warning signal accordingly. This warning does not necessarily mean an error has occurred at the output of the module. In the case of DWC the discrepancy may have occurred on the secondary copy that is not output from the module, but none the less, it creates a internal discrepancy and warning. For TMR the warning may not mean an erroneous output because the majority voting structure may correct the error before it appears at the output. When this warning signal is raised by the fault tolerant structures the time since the start of the injection campaign is recorded as a warning event for the test.

It is important to specify the name of the warning signal in the the generation process if one is present. If the name of a present warning signal is not included whenever a warning is generated it will be detected as an error. This detection of a warning as an error occurs because the module having faults injected into it will generate warnings but the gold standard module will not. This discrepancy in output states will be treated as an error by the error detection logic if the warning signal is not accounted for.

3.12.3 Recording an Injection Event

When the injection timer generates the signal to inject a fault into the module the same signal is also used to instruct the injection time memory to record the time since the fault injection campaign began. This recorded time corresponds to the time when the fault was injected into the module.

3.13 Unique Design Methodologies and Choices

This section will go over the design choices that were made, as it may not be immediately clear why some choices were made. Also this section will look at design methods that are unique to this implementation and the design trade-offs associated with these choices.

One unique part of the implementation is that during the injection campaign there is no need for any intervention by a host computer, all aspects of the injection campaign are contained within the FPGA. Not having a need for a host computer to control injection allows for the fault injection campaign to operate at the full circuit speed not requiring any slowdown for communication with a host. Using high speed communication interfaces does not alleviate this problem as the amount of data that has to be transferred in a very short time is the limitation. For a relatively simple design with a few hundred flops, at least one bit of data per flop has to be transferred to control each injection site. Even for reasonably slow circuit speed and a high data transfer speed there is not enough time within a single circuit clock to transfer all the necessary data to control the next injection. This time limitation results in the need to slow down or stop the circuit to allow for the data transfer to complete.

Allowing for the pre set up of the injection parameters and then allowing the fault injection tool to generate the rest of the fault injection locations and times provides a good balance between controlability, observability and speed. This ability to operate at the intended circuit speed is advantageous because for regulation purposes tests often need to be performed at the speed the device is intended to operate at. Also given a large enough sample data set it can be shown that tests can have statistical significance without having to test every error for every state of the circuit.

One design choice that may not be immediately clear is the operation used to range the flopID values described in section *3.7 Addressing the Flip-Flops*. Ranging the flopID value by dropping the MSB of the random value if it is not in the correct range creates an uneven distribution of values after ranging. There are other advantages to preforming the ranging this way other than it being hardware efficient design. Dropping the MSB of the random value if necessary allows for the ranging of the value into the correct range within a single clock cycle. This ranging is achieved using a very simple and resource efficient design compared to the commonly used ranging method of binning. The binning operation generates a random value then determines the corresponding ranged value using a comparison, it follows the form of a piecewise function. A simple example is given below to illustrate the concept where x is the random value and $f(x)$ is the ranged value.

$$f(x) = \begin{cases} 1 & : a \leq x \leq b \\ 2 & : b < x \leq c \\ 3 & : x > c \end{cases}$$

The binning operation is relatively efficiently implemented on a general purpose processor requiring only conditional statements and branching to implement. The equivalent hardware implementation that is able to preform the binning operation in a single clock cycle would require a large MUX or LUT to implement. For example the LFSR implemented in the design generates values on 63 bits, a MUX to decode this would require 63 select lines and 2^{63} inputs, and then the bit width of the flopID signal would have to be taken into account. Assuming the design had relatively few flops so the max flopID could be represented on 8 bits the resulting MUX would have

$8 * 2^{63}$ inputs. Attempting to do the ranging in this way would quickly consume all the available resources of the device resulting in an unusable design.

Another design choice that may not be immediately clear is the need to have injection periods that are $2^n - 1$ clock cycles long, where n is an integer between 1 and 32 as explained in section *3.8 Injection Timer*. Having injection intervals limited in this way does seem like a large drawback for the sake of a simplified design but there are advantages to the design. Limiting injection times to binary values that only contain 1s it is possible to guarantee a random value generated from a LFSR is within the correct range with a simple AND operation, also described in section *3.8 Injection Timer*. Other methods investigated allowing for arbitrary injection periods using a LFSR as the random number generator but there was no simple resource efficient method found that guarantied a properly ranged value within a one clock cycle window. Also the appearance of a limited range of injection periods can be overcome because an arbitrary number of faults can be injected within an injection period. The ability to inject any number of faults in an injection period allows for the the ability to create almost any injection rate. This is done by dividing the injection period by the number of injections that occur within that period.

3.14 Tool Application and Use Cases

The intended use for the tool is accelerated fault testing of FPGA based safety critical systems prior to deployment. The implemented design allows for run time configurable injection rates as well as the number of injections. The tool supports multiple simultaneous injections and can inject faults as fast as every single clock cycle. These abilities allow the tool to be used to perform a variety of tests and have a multitude of

applications. Given these abilities to perform unique tests it is still likely depending on the testing requirements multiple tests will have to be run to achieve statistically significant results.

The tool can quickly provide quantifiable results that can easily be converted into visual references. It also has the ability to test combinations of fault patterns and designs that current formal methods for determining fault tolerances would not be able to verify. The data generated by the tool can also be used to provide evidence to customers or regulating bodies that a design meets the specified fault tolerance they require. If the consumer or regulator proposed a new test case the tool can be used to quickly provide a preliminary analysis of the new test case. The tool can also be used to perform comparison tests of different designs vs the same fault rates and chip area vs design execution time comparisons. For example if it is known that for a given amount of chip area n number of faults will occur per second an investigation can be performed to see if a design that uses 3 times the chip area but runs in 1/3 the time is more prone to errors then a design that uses 1/3 the area but requires 3 times longer to run. Further tests could compare the fault recovery rate of different fault tolerant structures or variations of ways to implement the same redundant structure. The methods outlined above just some possible uses of the tool, a tester can use the fault injection features that are provided in order to better asses the error resilience of the system at hand.

Chapter 4

Experiments Performed

To validate the operation of the fault injection tool and generate preliminary data, the fault injection tool was tested on a *luminance (Y) chrominance (UV)* interpolator as part of a simplified JPEG decoder. YUV is a representation of the red green and blue colour space often used in image compression. In the case of this implementation the compression uses 1/4 the sample depth to represent the U and V components then it does to represent the Y, i.e. the Y values are sampled at 32 bits and the U and V values are sampled at 8 bits. The interpolator performs the upsampling of the U and V components to match the depth of the Y component. The interpolation is done using an 8 bit wide 6 tap filter that multiplexes filter taps based on the state. Three versions of the interpolator were tested: a regular non-redundant version, a version modified to operate with DWC and a version modified to operate with TMR. Using these three structures allows for preliminary investigation into the effects of fault injection on fault tolerant designs as well as providing a testing platform for the tool.

In this chapter unless otherwise specified when referring to something as random

it will actually refer to pseudo-random.

The terms “fault”, “injection” or “injected fault” will refer to a simulated SEU before any effect can be observed at an output and are used interchangeably.

The term “error” will exclusively refer to an erroneous value at an observable output.

4.1 Design of the fault tolerant structures

Modification were made to the interpolator to enable operation with fault tolerant structures. This section briefly outlines how the modifications were made and the basic functionality of the modifications.

4.1.1 Design of the DWC Interpolator

For the DWC implementation the internal structure of the interpolator is duplicated and each replica provided the same inputs. DWC does not provide any fault recovery mechanism, only fault detection, so only the output from one of the duplicated modules is passed back to the higher level module. The copy that passes the output back to the higher level module we will call the primary copy and the copy used for comparison will be called the secondary copy. The comparison that generates the error warning signal is only performed on the outputs of the module not on the full state of all the registers. An internal discrepancy warning is produced if there is any difference between any of the corresponding primary and secondary output values. Figure 4.1 illustrates the DWC design.

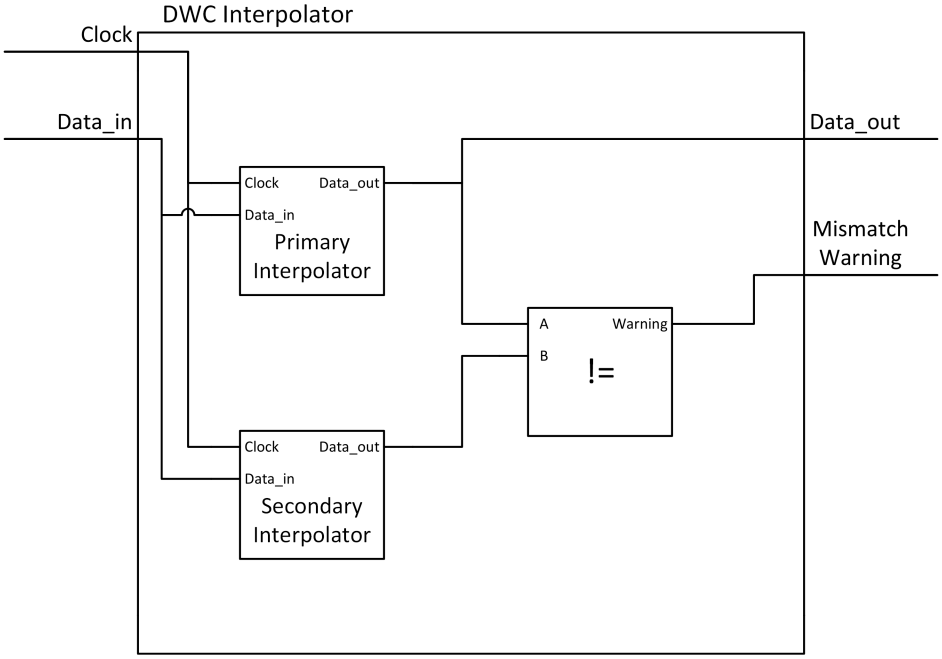


Figure 4.1: Diagram of DWC Implementation

4.1.2 Design of the TMR Interpolator

The implementation of the TMR interpolator triplicates the internal structure of the interpolator. Each copy is provided the same inputs from the higher level module and each copy produces an individual output. The outputs from each module are then provided to a majority voter to produce the final module output. The TMR voting is only performed on the module outputs, not on the state of all the registers in the circuit. The TMR voting is based on a word based comparison not a bitwise majority vote. If 2 corresponding outputs match that value will be chosen as the correct value in the case that all 3 copies produce different results the primary copies value is output by default default. If any of the outputs do not match either of the corresponding copies outputs, an internal discrepancy warning will be generated. Figure 4.2 illustrates the design of the TMR interpolator.

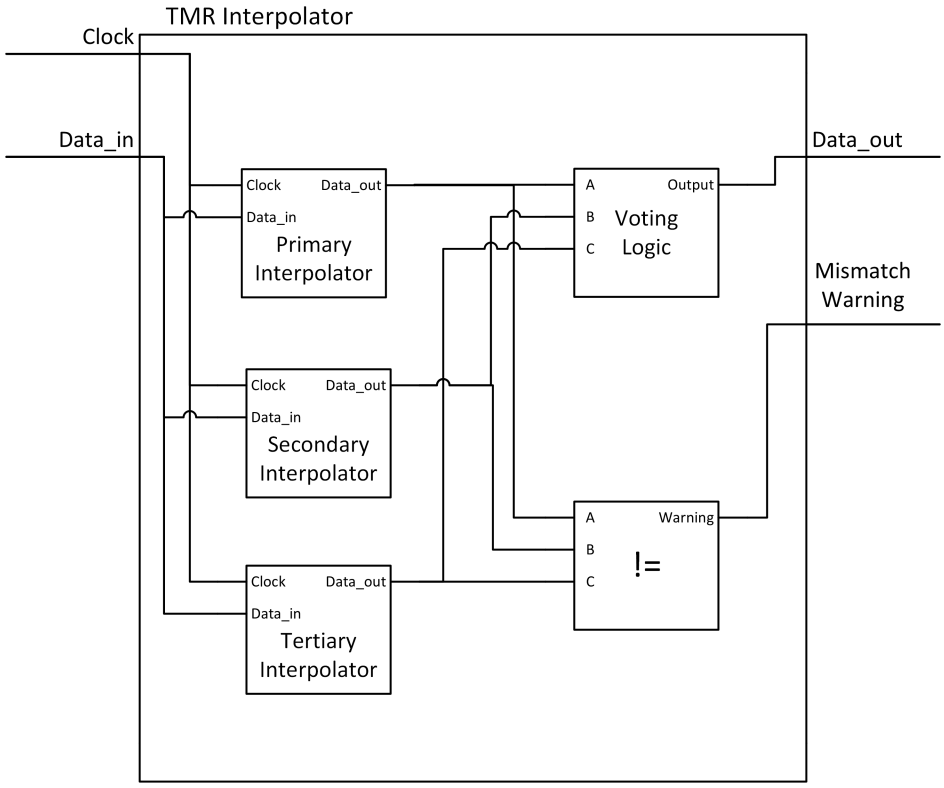


Figure 4.2: Diagram of TMR Implementation

4.2 Tool Growth Rate and Area Estimation

This section will examine the rate at which the tool uses resources compared to the size of the original design it is modifying. This is to determine the growth rate of the resources consumed by the tool.

Examining a design with k flops supporting m injections and n-bits of output data. The tool represents the number of flops in the design in binary requiring $\lceil log_2 k \rceil$-bits for that representation. The number of gates required for the one-hot decoders is then $(m * 2^{\lceil log_2 k \rceil} * (\lceil log_2 k \rceil - 1))$ and the number of ORs needed for the multi-hot decoder is $((m-1) * 2^{\lceil log_2 k \rceil})$. The number of gates required for the error detection comparison is $(2 * n - 1)$, the number of registers required for the LFSRs is $(m * 2 * 63)$, and the

other ancillary registers require $(m * (63 + 32 + 1 + \lceil log_2 k \rceil + \lceil log_2 m \rceil) + 36 + 64)$ registers.

From the growth rates above the greatest are the rates for one-hot decoders and the ORs for the multi-hot decoder. Both of those rates are on the order of $2^{\lceil log_2 k \rceil}$. This exponential simplifies to k meaning overall the growth rate for the tool is linear.

The growth rate is also illustrated in the graph in figure 4.3. The graph plots the total LEs used in the design not the total number of registers because the majority of the tool's resource usage is combinational logic and does not use registers. The design of the non-redundant structure has 80 flops, the DWC structure has 160 flops and the TMR structure has 240 flops, and all three designs have 40-bits of output data. For the base designs the number of registers register reported by the compiler is 165, 245 and 325 for non-redundant, DWC and TMR respectively. The reported numbers show an increase of the expected 80 gates for each design. Other control logic is generated in the base design number of register and LEs used greater, further the fault tolerant structures in the DWC and TMR add more logic as well.

Reading the individual plots in figure 4.3 it can be seen that the initial growth occurs when the first injector is added. As subsequent injectors are added the number of resources consumed increases at a lower rate. Looking a the graph vertically it can be seen that when the first injector is added the growth from non-redundant to DWC is less than the growth from DWC to TMR. For a linear growth rate these differences should be approximately equal. The larger final size for the TMR structure can be explained by the extra voting logic that TMR contains as opposed to the other two designs. Additionally if the tool did not have a linear growth rate the DWC and TMR designs would have much higher LE usage.

A rough estimation of the resources used by the tool can be done using the sum of the equations stated above. If the number of flops in the design, k, is closer to the maximum number that $\lceil log_2 k \rceil$-bits can represent the equation tends to underestimate the resource usage. The equation tends to overestimate resource usage for designs where k is further from the maximum number that $\lceil log_2 k \rceil$-bits can represent. One reason for this is it doesn't take into account the FSM logic and some of the other counters. Another reason is the compiler performs optimization on the design so logic is reduced when it is not needed. The result of the optimization is the ceiling function in the estimation tends to cause overestimation of the smaller designs.

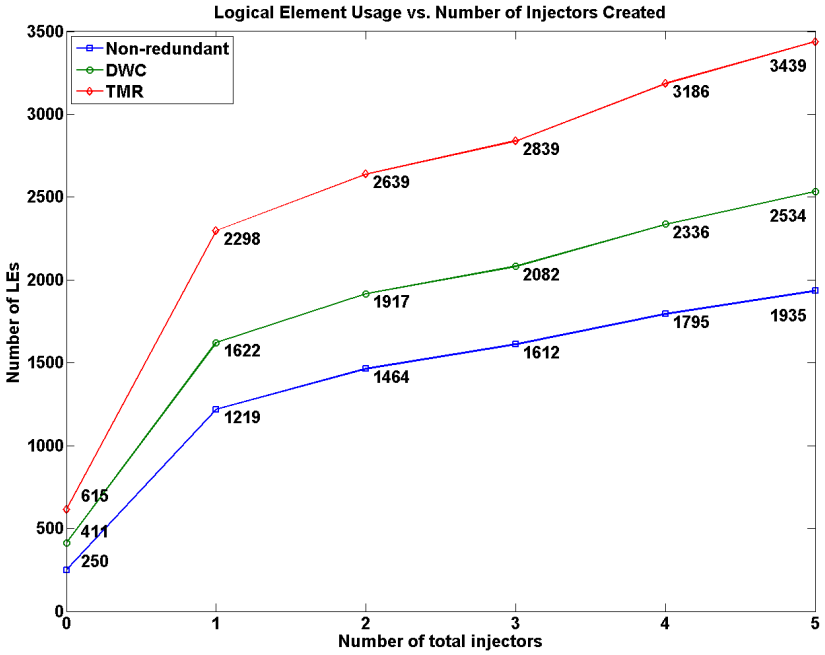


Figure 4.3: Logical Element Usage vs. Number of Injectors Created

4.3 LFSR Subset Distribution Analysis

The values generated by a maximal LFSR have a statistical even distribution, but this distribution is over the whole period of the LFSR values. The tool as implemented only uses a subset of the bits from the LFSR generated value to produce a random value. This means that it is possible for the distribution of values generated from the LFSR subset to be non-uniform. If the resultant distribution was non-uniform it would skew the results and would have to be taken into account in the analysis.

This section will look at some samples of LFSR subset values to determine if the values are evenly distributed. The LFSR examined is the same one used in the implementation described in section 3.6 with the characteristic polynomial of $P(x) = x^{63} + x^{62} + 1$.

40 Samples of 2048 values were generated on the FPGA each using a unique random seed value. 20 of these samples are the 8 least significant bits of the LFSR value and the other 20 are the 32 least significant bits of the LFSR. The reason 8 and 32 are the chosen bit widths is because they are the same bit widths used during the testing of the interpolators. 8 bits is the width used for the random flop selection and 32 bits is the width used for the random injection timer.

Table 4.1 lists the sample means and standard deviations of the 8-bit and 32-bit samples, also the table includes the mean and standard deviation of a cumulative sample consisting of all of the samples for the respective bit widths. For an ideal uniform distribution the mean can be calculated using equation 4.1 and the standard deviation can be calculated using equation 4.2, where; A is the lowest possible value in the distribution, and B is the greatest possible value in the distribution (Montgomery and Runger, 2007).

$$A + B/2 \tag{4.1}$$

$$\sqrt{(B-A)^2)}/12 \tag{4.2}$$

Table 4.2 lists theses ideal values along with the greatest percent deviation between the sample means and sample standard deviation, and the percent deviation of the cumulative sample mean and standard deviation. The greatest difference between the individual samples and the ideal value is less than 5%, and the histograms of the samples in figures 4.4 and 4.5 are not skewed in any particular way suggesting the distributions are even. Both of the cumulative sample values differ by less than 1% from the ideal values suggesting that repeated tests with different LFSR seed values will result in even distributions.

Sample Number		8-bit Samples			32-bit Samples	
		Sample Mean	Sample Standard Deviation		Sample Mean	Sample Standard Deviation
1		125.51	75.31		2.0872e+09	1.2659e+09
2		126.73	75.05		2.1331e+09	1.2410e+09
3		125.81	74.64		2.1260e+09	1.2450e+09
4		129.32	73.80		2.1028e+09	1.2538e+09
5		125.42	74.38		2.1077e+09	1.2495e+09
6		127.21	74.03		2.1009e+09	1.2506e+09
7		127.06	74.36		2.1631e+09	1.2348e+09
8		129.79	72.17		2.1077e+09	1.2530e+09
9		127.85	74.22		2.2113e+09	1.2050e+09
10		124.45	74.93		2.2195e+09	1.2053e+09
11		126.06	74.90		2.1304e+09	1.2441e+09
12		132.88	71.63		2.0525e+09	1.2790e+09
13		129.06	74.07		2.1778e+09	1.2251e+09
14		127.59	74.30		2.1620e+09	1.2341e+09
15		129.29	73.02		2.1096e+09	1.2792e+09
16		125.06	75.48		2.1620e+09	1.2398e+09
17		127.65	73.97		2.1515e+09	1.2316e+09
18		128.17	74.02		2.1822e+09	1.2257e+09
19		130.76	72.08		2.1237e+09	1.2396e+09
20		123.22	76.27		2.1921e+09	1.2303e+09
Cumulative		127.44	74.16		2.1402e+09	1.2422e+09

Table 4.1: Sample Means and Standard Deviations

		8-bit			32-bit	
		Mean	Standard Deviation		Mean	Standard Deviation
Ideal Values		127.50	73.61		2.1475e+09	1.2399e+09
Greatest Deviation in Samples from Ideal		4.22%	3.61%		4.4222%	3.1707%
Cumulative Sample Deviation from Ideal		0.04%	0.74%		0.3412%	0.1894%

Table 4.2: Percent Difference from Ideal Mean and Standard Deviation

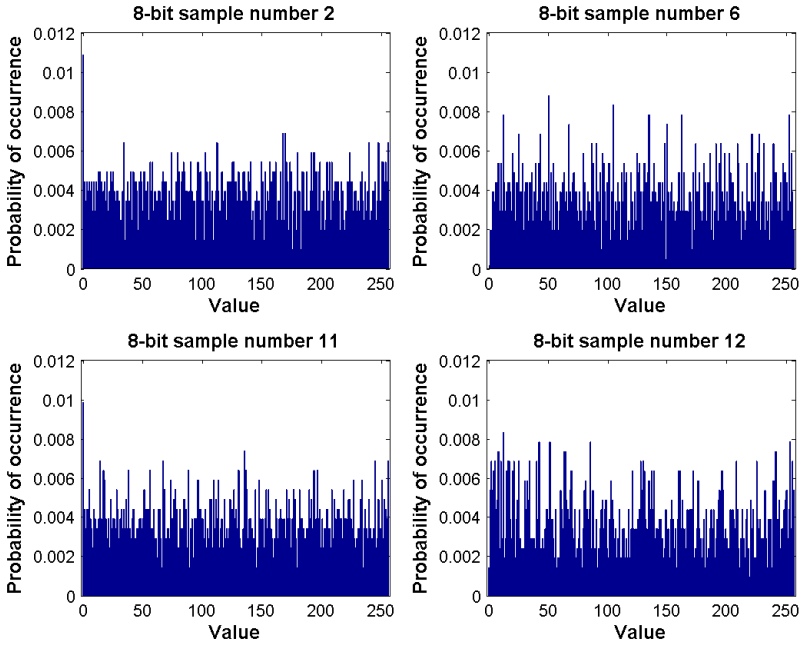


Figure 4.4: Histograms of 4 of the 20 8-bit samples

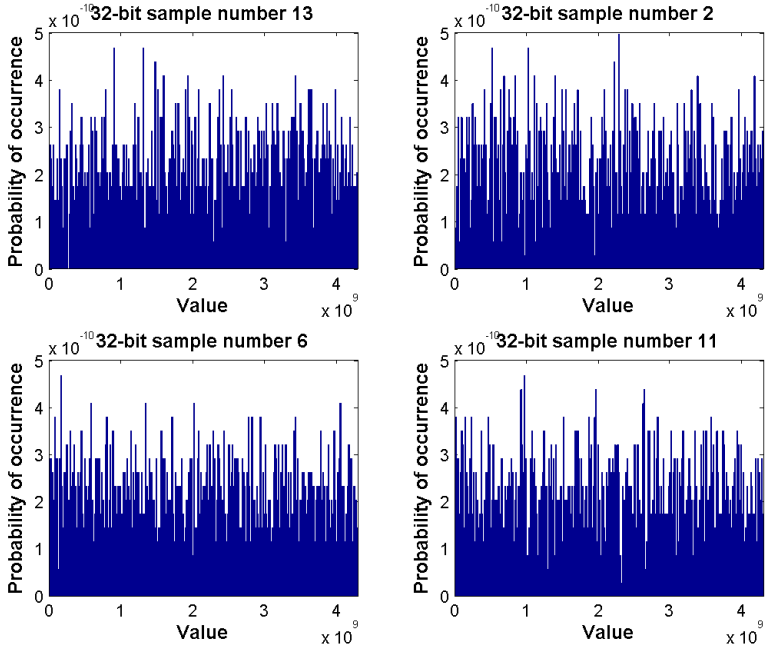


Figure 4.5: Histograms of 4 of the 20 32-bit samples

4.4 Tests Performed

Five different tests were performed on each version of the interpolator to evaluate the effects of fault injection on the different structures. The injection period (t_{IP}) of all the tests is $2^{20} - 1$ clock cycles. This section outlines the parameters of each of the tests.

4.4.1 Test 1i - Single Injection Test

The Single injection test injects one fault into a random flop location within the circuit being tested at a random time within the injection period. This is repeated until test completion.

4.4.2 Test 2i - Two Injection Test

For this test one fault is injected into a random flop location within the circuit being tested at 2 random times within the injection period, this results in an average injection period of $t_{IP}/2$ clock cycles.

4.4.3 Test 2i - Three Injection Test

This test injects one fault into a random flop location within the circuit being tested at 3 random times within the injection period, this results in an average injection period of $t_{IP}/3$ clock cycles.

4.4.4 Test 2s - Dual Simultaneous Injection Test

This test injects two faults into the structure within the same clock cycle. The two faults are injected into two random flop locations within circuit being tested. The simultaneous injections are performed at a random time within the injection period, but result in an effective fault period of $t_{IP}/2$ clock cycles because two faults are being injected at a time.

4.4.5 Test 3s - Triple Simultaneous Injection Test

This test injects three faults into the structure within the same clock cycle. The three faults are injected into three random flop locations within circuit being tested. The simultaneous injections are performed at a random time within the injection period, but result in an effective fault period of $t_{IP}/3$ clock cycles because three faults are being injected at a time.

4.5 Data Recorded During the Test

During the fault injection tests a timer counts the number of clock cycles that the fault injection tool has been running for. This timer allows for the recording of times when specific events have occurred during the test. The events that are recorded are: the time when a fault is injected, the time an erroneous output is detected, and, in the case of fault tolerant structures, when a warning is generated.

4.6 Structure of Memories During Tests

Each of the memories for testing were instantiated with 2048 locations, each location being 48 bits wide. These sizes were chosen to maximize the amount of recorded data within the confines of the test platform and to guarantee a sufficiently long operating period. The width of 48 bits allows for an operating period of approximately 65 days at an operating frequency of 50MHz.

For the tests described in this thesis, each test is run until either all the memories are full or 20 minutes has elapsed. During the test each memory location is only written to once. When the end of the memory is reached, writing to that memory is stopped for the remainder of the test. The rate that each memory fills at is dependent on the frequency of the event that the memory is recording. These varying rates results in writing to each memory being halted at separate times.

Allowing the memories to halt recording individually rather than stopping all recording when the first memory is full enables the collecting of more samples than if recording was halted earlier. These samples are used to calculate average rates of event occurrences, thus, more samples result in more accurate averages.

4.7 Definition of an Error Event

An injected fault may not affect the outputs of the circuit it is injected into or it may take many clock cycles for an error to propagate to an output. Additionally a single injected fault may result in multiple errors, and these errors may occur over multiple clock cycles.

Figure 4.6 shows a single injected fault causing errors over multiple clock cycles. In

the signal "Inject Error" is generated when a fault is injected and the signal "Output Error" is generated when an error is detected at the output of the circuit.

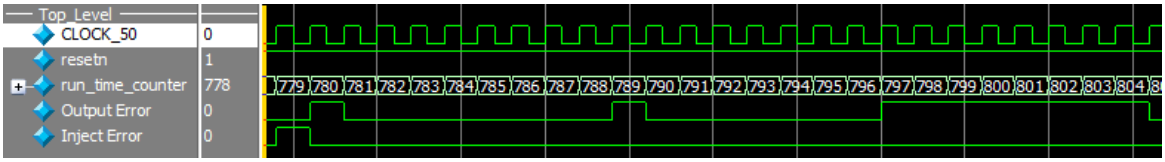


Figure 4.6: Example of a single injection causing multiple errors

The two major metrics looked at when analyzing error events are how long after an injection an error is detected and the other metric is the rate at which error events occur. Defining an error event simply as when the "Output Error" signal is high is incorrect for the analysis performed due to the potential of a single injection causing errors over multiple clock cycles. If every time when the error signal was high was used to calculate the error rates this would greatly inflate the error rate over the injection rate. Additionally if the first error after an injection is taken as to have been caused by that injection it could lead to the interpretation that some faults take a very long time to propagate into an error when in actuality the error detected was provoked by a subsequent injection. The data could be misinterpreted, having the appearance that the injection to detection times are greater than would be possible given the circuit design.

4.7.1 Error Detection Times

For the analysis an error is defined to be the first detection of an erroneous output within 50 clock cycles after an injection has been made. Meaning if an injected fault does not produce an error within 50 clock cycles it is defined to have not generated an error for the purposes of the analysis. A 50 clock cycle limit was chosen because given

the circuit it is enough time to allow any injected fault to propagate to an output if it is going to. For other circuits a limit of 50 clock cycles may not be an appropriate the length of time. The maximum propagation time would have to be determined individually for each circuit being tested and taken into account in the analysis. The same analysis was also performed with a limit of 500 and 1000 clock cycles, and the results of the analyses were unchanged.

4.8 Description of Tables and Graphs

This section will go over the structure of the tables and graphs used in the analysis of the test results.

4.8.1 Clock Cycles Till Detected Error Tables

Tables 4.3, 4.5, 4.7 contain an analysis of how long after an injection an error is first detected. Below is a brief description of what each column in each table represents.

Test Run - This column specifies which test was run. The data in the corresponding row was generated from the test that is specified by this column.

Average Injection Rate - This column specifies the calculated average rate that injections were made at. This rate is based on the recorded injection times.

Apparent Fault Rate - This column specifies the calculated average rate that faults are injected into the module being tested at. For non-simultaneous injections this rate is the same as the average injection rate. For simultaneous injections this rate is the average rate of injection multiplied by the number of simultaneous injections.

No Error Detected - This column specifies the percent of total faults injected that did not propagate to an observable error.

Error after "n" Cycle(s) - This column specifies the percentage of total injected faults that propagate to an observable error that is first detected "n" cycles after it is injected.

4.8.2 Calculated Rate Tables

Tables 4.4, 4.6, 4.8 tabulate the average rates of event occurrences. The first 3 columns test run, average injection rate and apparent fault rate are identical to those described in section 4.8.1. The new columns are described below. Additionally for the tests run on the DWC and TMR structures an average warning rate is also present.

Average Erroneous Output Rate - This column specifies the average rate that errors were detected. The rate is calculated based on the time when an error is first detected after an injection.

Average Warning Rate - This column specifies the average rate that the fault tolerant designs generate warnings. The rate is calculated based on the times after an injection a warning is first recorded.

4.8.3 Graphs

The graphs in the following sections are histograms of how many clock cycles after an injection an error was first detected.

4.9 Non-Redundant Structures Tests Results and Analysis

Table 4.3 tabulates how long after the injection of faults errors were detected for the non-redundant structure. For the non-simultaneous injections an average of only 36.32% of injections result in an error propagating to an output. This low ratio of error propagation means the design of the interpolator is naturally resistant to this type of fault injection. Errors that are detected predominantly occur one clock cycle after injection, this is illustrated in the histograms in figures 4.7 and 4.8. An average of 29.74% of faults propagate to an output one clock cycle after injection. The number of injections that result in errors first detected more than one clock cycle after injection are far less frequent, cumulatively making up only 6.57% on average. Increasing the injection rate for non-simultaneous injections does not appear to have any observable effect on the quantity of faults that propagate into errors.

Simultaneous fault injection greatly increases the percentage of injected faults that propagate to the output as errors. The percent of errors detected one clock cycle after injection for test 2s is 47.78% and for 3s is 64.29%. This is a significant increase over the average detection after one clock cycle of 29.74% for non-simultaneous injections. Although simultaneous injections increase the number of faults that propagate into errors it does not appear to increase the number of times an error will first occur more than one clock cycle after injection, as errors are still predominantly observed one clock cycle after injection.

Table 4.4 displays the average injection and fault rates for the non-redundant

structure. As expected, with non-simultaneous injections as the injection rate increases the rate at which errors occur also increases. Similarly for simultaneous injections as the number of simultaneous injections increase the average error rate also increases.

Test Run	Average Injection Rate [injections/s]	Apparent Fault Rate [faults/s]	No error detected [%]	error after 1 cycle [%]	error after 2 cycle [%]	error after 3 cycle [%]	error after 4 cycle [%]	error after > 4 cycle [%]
1i	47.60	47.60	63.95	29.24	2.80	2.80	0.40	0.80
2i	95.25	95.25	63.53	29.17	2.81	3.37	0.56	0.56
3i	138.55	138.55	63.56	30.82	2.33	1.64	0.55	1.10
2s	47.70	95.39	46.44	47.78	3.78	0.44	0.00	1.56
3s	47.68	143.05	31.59	64.29	2.47	0.82	0.27	0.55

Table 4.3: Non-Redundant structure clock cycles till detected error

Test Run	Average Injection Rate [injections/s]	Apparent Fault Rate [faults/s]	Average Erroneous Output Rate [errors/s]
1i	47.60	47.60	17.12
2i	95.25	95.25	34.58
3i	138.55	138.55	48.96
2s	47.70	95.39	25.61
3s	47.68	143.05	32.61

Table 4.4: Non-Redundant structure rates

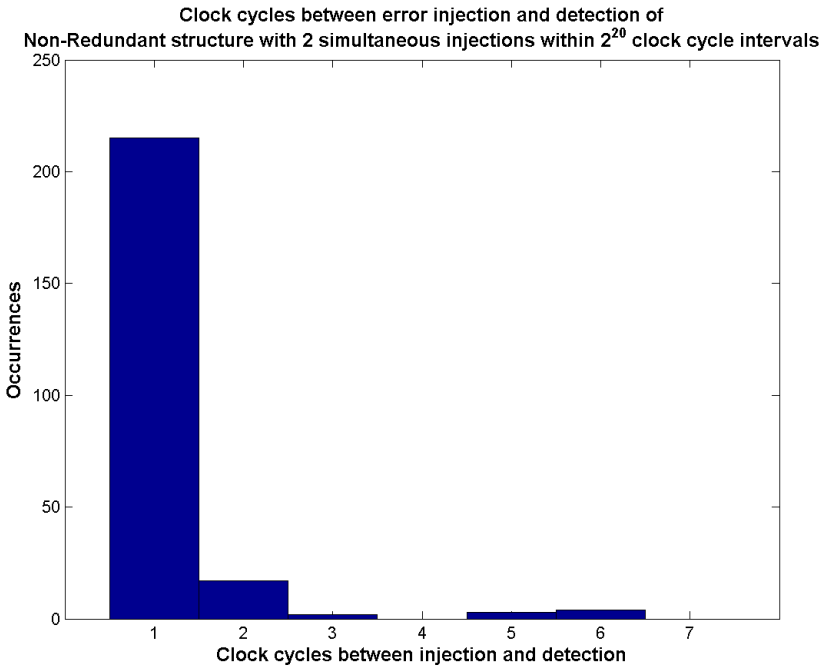


Figure 4.7: Histogram of Non-Redundant structure with 2 simultaneous injections within t_{IP}

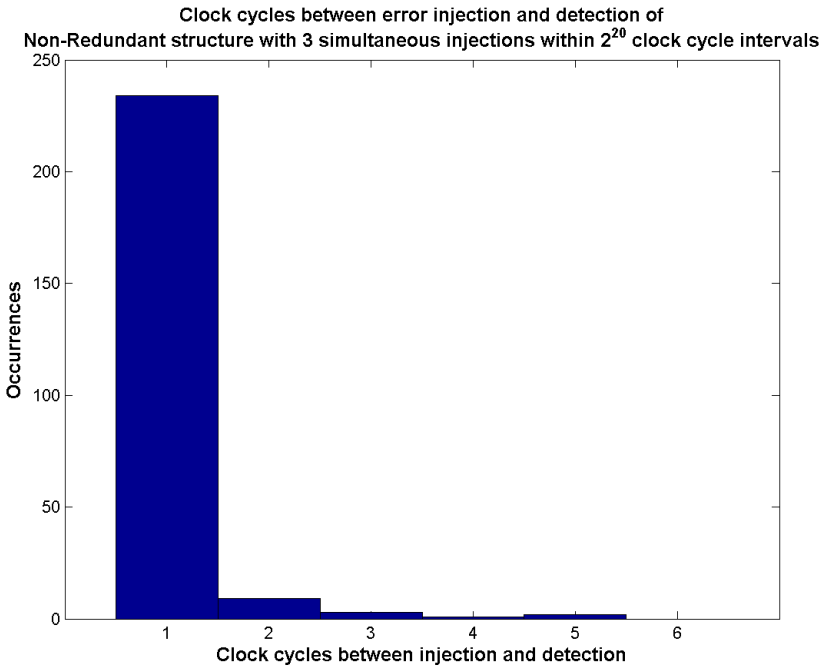


Figure 4.8: Histogram of Non-Redundant structure with 3 simultaneous injections within t_{IP}

4.10 DWC Test Results and Analysis

The DWC structure follows approximately the same pattern as the non-redundant structure. For the single injection tests increasing the injection rate does not change the probability a fault propagates into an error. Increasing the number of simultaneous injections increases the probability and error is observed. The majority of observed errors occur one clock cycle after injection, and the number of errors first observed more than one clock cycle after an injection sharply falls off, this is illustrated in the histograms in figures 4.10 and 4.11.

The DWC structure contains exactly twice the number of flops as the non-redundant structure but only half of the flops are used to generate an output, the other half are are only used to generate a comparison. Thus for the same fault rate the number of errors detected for the DWC structure should be approximately half the number detected for the non-redundant structure, and the rate errors occur at should also be halved. The rate at which the DWC structure generates warnings should then be equal to the error rates for the non-redundant structure, given the same fault rates.

The number of occurrences and the rates in tables 4.5 and 4.6 partially support this. The number of occurrences and the rates for the DWC structure are lower than they are for the non-redundant structure, but they are not halved as would be expected. For non-simultaneous injections the total percent of detected faults drops from an average of 36.32% for the non-redundant structure to an average of 23.56% for the DWC structure. This drop is a reduction in the number of faults that propagate to errors by approximately 35%. For the simultaneous injection tests the drop in the number of faults that propagate to errors is approximately 23.31%. The drop in detections in both the simultaneous and non-simultaneous tests is far

from the expected 50% drop. The erroneous output rates drop similarly, for the non-simultaneous tests the rate drops by approximately 34% and for the simultaneous injection tests the rates drop by approximately 22.9%.

Test Run	Average Injection Rate [injections/s]	Apparent Fault Rate [faults/s]	No error detected [%]	error after 1 cycle [%]	error after 2 cycle [%]	error after 3 cycle [%]	error after 4 cycle [%]	error after > 4 cycle [%]
1i	47.23	47.23	76.76	20.60	1.17	0.59	0.00	0.88
2i	95.05	95.05	76.53	20.48	1.38	0.66	0.22	0.73
3i	142.74	142.74	76.15	20.89	1.30	0.94	0.36	0.36
2s	47.69	95.39	57.59	38.89	1.22	0.81	0.81	0.68
3s	47.69	143.07	49.24	47.86	1.22	0.61	0.61	0.46

Table 4.5: DWC structure clock cycles till detected error

Test Run	Average Injection Rate [injections/s]	Apparent fault Rate [faults/s]	Average Erroneous Output Rate [errors/s]	Average Warning rate [detections/s]
1i	47.23	47.23	10.94	17.00
2i	95.05	95.05	22.32	38.38
3i	142.74	142.74	34.04	55.26
2s	47.69	95.39	20.21	28.50
3s	47.69	143.07	24.21	31.71

Table 4.6: DWC structure rates

Originally a potential explanation for this lower than expected drop in error propagation was believed to be the uneven distribution of register selection due to the randomly generated values having to be ranged into the number of flops in the circuit being tested.

For the DWC structure the possible flopID values are [1-160] but the possible LFSR values are [1-255], so values [161-255] are remapped into the valid range. The way the remapping currently operates the values [161-255] are directly remapped to [35-127]. Figure 4.9 shows how given an even distribution before remapping results in an uneven distribution after remapping. Similarly for the non-redundant structure the LFSR generated values [81-127] are remapped to [17-63].

If all the flops in the design have the same probability that injecting a fault into them will cause an error this uneven distribution would not cause a problem, but this is not the case. For both the DWC structure and the non-redundant structure the flops with flopIDs [49-80] are directly connected to an output so a fault injected into one of those locations will always cause an error in the next clock cycle. Again for both structures flops [1-48] may result in an error if a fault is injected into one of them, but it is dependent on the current state of the circuit when the fault is injected. Flops [81-160] are only present in the DWC structure and, will not result in a delectable error if a fault is injected into them because those flops are only used to generate the value used for comparison.

The main difference between the redistribution of the DWC structure vs the non-redundant structure that was thought to be causing the discrepancy from the expected 50% drop are the probabilities that a sensitive flop is targeted. For the non-redundant

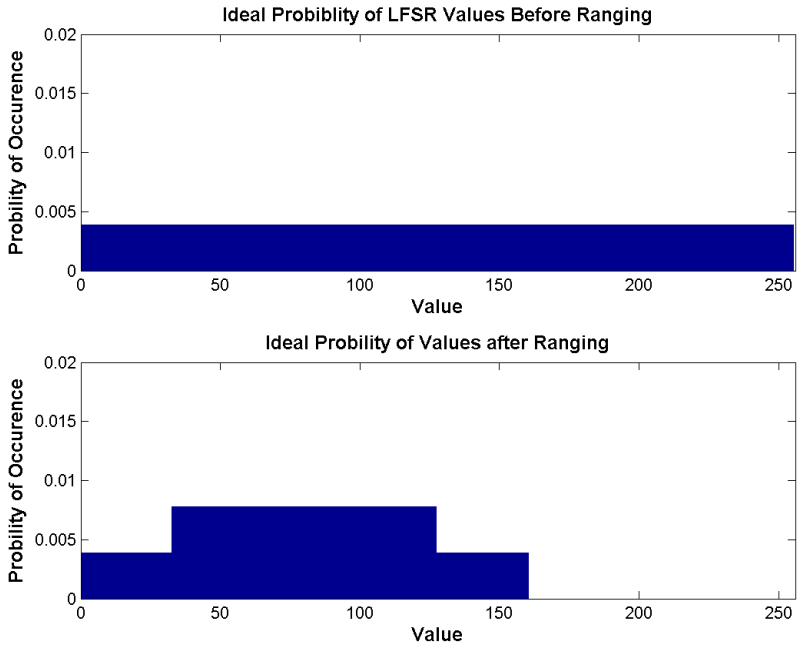


Figure 4.9: Remapping Causing Uneven Distribution

structure there are 47 values that will target a sensitive flop, but for the DWC structure there are 64 values that will target a sensitive flop, but there are only 127 valid LFSR values for the non-redundant structure vs 255 for the DWC structure. This mean there is a 37% chance that a sensitive flop will be targeted in the non-redundant structure, but only a 25% chance a sensitive flop will be targeted in the DWC structure.

This result means that the expected 50% drop in error events at minimum should have been observed. Thus either the sample size is too small and this is an outlying case or further investigation into the structure is needed to determine why the experimental results differ from what is expected.

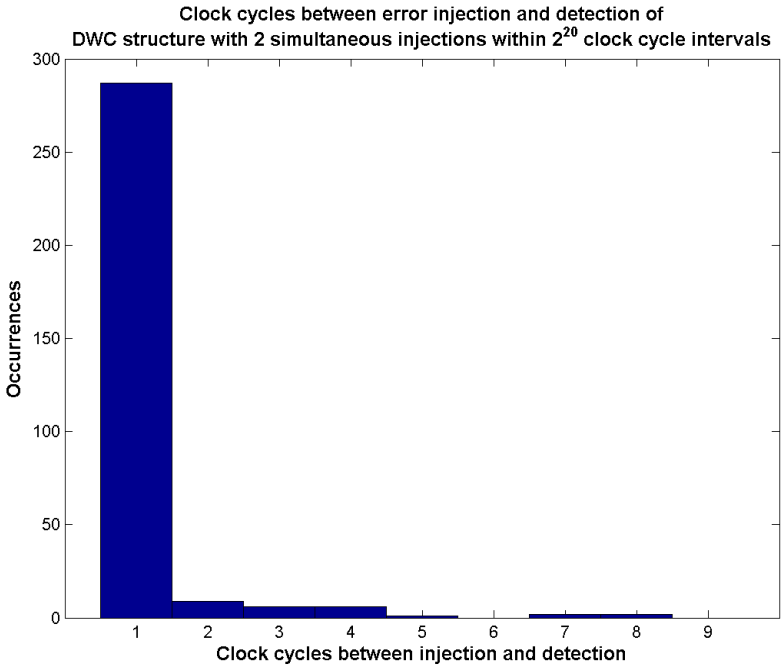


Figure 4.10: Histogram of DWC structure with 2 simultaneous injections within t_{IP}

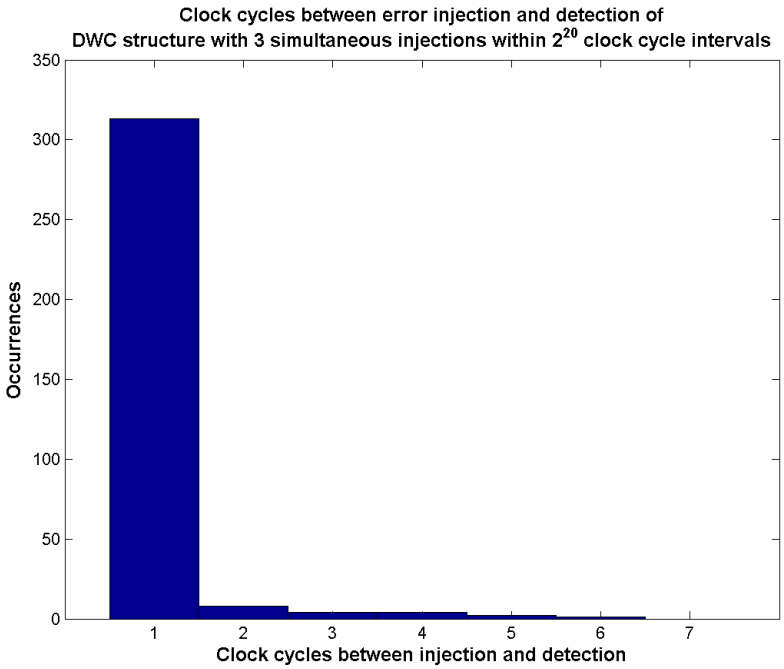


Figure 4.11: Histogram of DWC structure with 3 simultaneous injections within t_{IP}

4.11 TMR Test Results and Analysis

For the non-simultaneous tests TMR, as expected, prevents any faults from propagating into errors. As a result for tests 1i, 2i and 3i no errors are detected in the 20 min the tests are run for.

The simultaneous injection tests create conditions where injected faults do propagate into errors. Distinctly from the non-redundant structure and the DWC structure results the number of errors detected one clock cycle after injection does not dominate the number of errors detected for the TMR structure. The distribution of how long after an injection an error is detected is still weighted towards 1 clock cycle after an injection, as can bee seen in the histograms in figures 4.13 and 4.14. Additionally in figures 4.13 and 4.14 it can be seen that there are errors that are first detected up to 29 clock cycles after an injection. This result could indicate that there are errors that would appear sooner in the other structures but TMR is able to recover from the simpler effects of the error, only failing on more complex consequences of the injection. In other words the injection could cause an error that would cause other structures to fail immediately and continually have errors over multiple clock cycles but the TMR structure is to recover from many of the preliminary errors only failing on the latter ones. Overall number of errors that occurred were greatly for the TMR structure, for tests 2s only 11.47% of injections resulted in errors, and for test 3s only 26.08% of injections resulted in errors.

Looking at table 4.8 for the first 3 tests the error rates are 0 errors/s because no errors were detected over the course of the test. For test 2s and 3s the error rates are relatively consistent with the proportion of injections that resulted in errors, the error rates are 8.03% and 15.96% of the injection rates for tests 2s and 3s respectively.

```
0b11101001 x 0b10011111 = 0b1001000010110111
   (233    x     159    =        37047)

0b11001001 x 0b10011111 = 0b0111110011010111
   (201    x     159    =        31959)
```

Figure 4.12: Example of How a Single Fault Can be Magnified

The warning rates generated by the TMR interpolator are very consistent with the warning rates for the DWC structure in table 4.6 and the error rates for the non-redundant structure in table 4.4, each set being very similar.

The faults that are propagating past the TMR voter and being detected as errors are likely faults that are being injected into flops that are being used in a multiplication. When a single error is injected into a multi-bit number the value that results from a multiplication with that erroneous number may have many bits that are incorrect. Figure 4.12 illustrates how a single fault can be magnified after a multiplication into 7 faults. This magnification of faults is likely what causes the TMR voter to vote for an incorrect output. After values are multiplied they are put into an accumulator before being output, this accumulator is likely where faults linger for some time and cause errors. If the accumulators in more than one copy of the TMR structure is corrupted by fault magnification the majority voter would choose the erroneous value if it was the majority, resulting in an error propagating to the output.

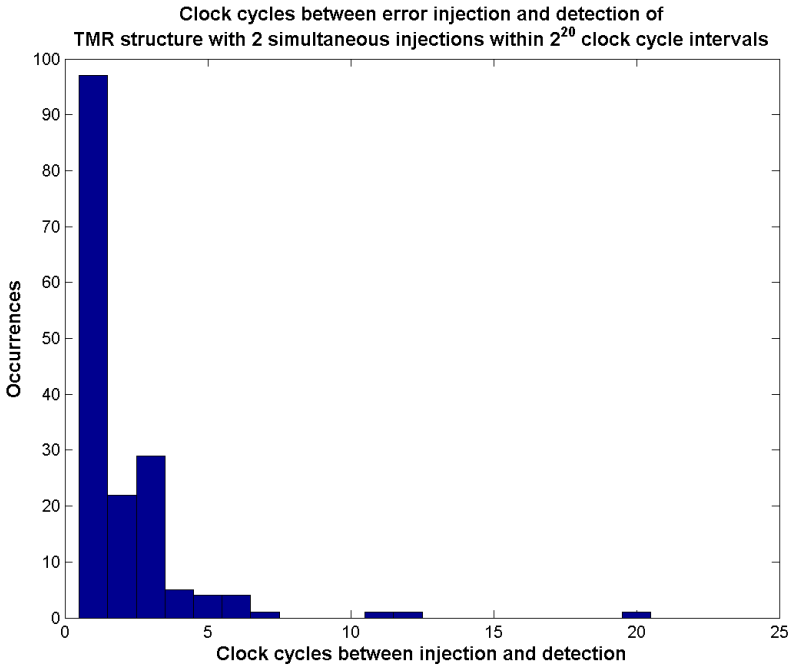


Figure 4.13: Histogram of TMR structure with 2 simultaneous injections within t_{IP}

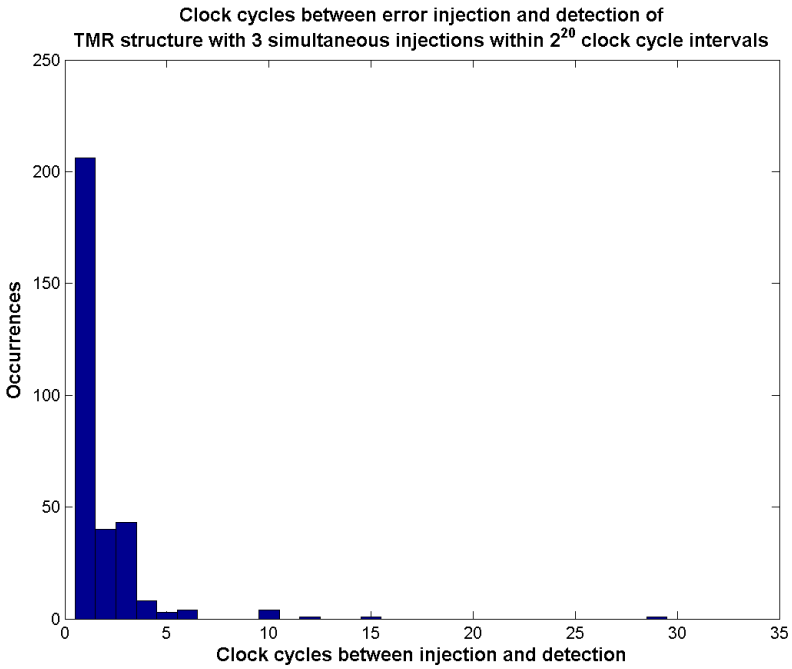


Figure 4.14: Histogram of TMR structure with 3 simultaneous injections within t_{IP}

Test Run	Average Injection Rate [injections/s]	Apparent Fault Rate [faults/s]	No error detected [%]	error after 1 cycle [%]	error after 2 cycle [%]	error after 3 cycle [%]	error after 4 cycle [%]	error after > 4 cycle [%]
1i	47.34	47.34	100.00	0.00	0.00	0.00	0.00	0.00
2i	95.42	95.42	100.00	0.00	0.00	0.00	0.00	0.00
3i	141.62	141.62	100.00	0.00	0.00	0.00	0.00	0.00
2s	47.71	95.41	88.53	5.76	1.76	2.29	0.59	1.07
3s	47.70	143.09	73.93	18.84	3.04	2.59	0.54	1.07

Table 4.7: TMR structure clock cycles till detected error

Test Run	Average Injection Rate [injections/s]	Apparent fault Rate [faults/s]	Average Erroneous Output Rate [errors/s]	Average Warning rate [detections/s]
1i	47.34	47.34	0.00	17.39
2i	95.42	95.42	0.00	37.06
3i	141.62	141.62	0.00	52.79
2s	47.69	95.39	3.83	27.54
3s	47.67	143.02	7.61	33.11

Table 4.8: TMR structure rates

Chapter 5

Conclusion and Future Work

The tool described in this thesis is design time parameterizable and runtime configurable. As implemented the tool requires minimal communication from a host computer allowing for the reduction of test execution time compared to other proposed methods. The reduction in runtime is achieved by removing the overhead associated with having a host computer to control the injection campaign. The elimination of the host computer also allows for the ability to inject faults as fast as every single clock cycle without any slowdown to the operating speed of the circuit. This full speed operation satisfies regulation and the ability to inject errors every single clock cycle is also a nice feature to have if a worst case scenario test is needed to be performed.

There are limitations to the tool including the distribution of pseudo-random values and the fixed intervals for the injection periods. These limitations mostly stem from the need to be able to generate a ranged random number within a single clock cycle.

The results from the test indicate that the tool's injection pattern has good coverage. This coverage can be seen from the error and warning rates in tables 4.4, 4.6,

and 4.8. The error rates for the non-redundant structure, the warning rates for the DWC structure, and the warning rates for the TMR structure are all approximately equivalent for equivalent fault rates. This result is to be expected as the warning rates for the redundant structures are equivalent to the number of errors that would have been generated given a non-redundant structure. The result shows that for different test structures the tool operates with equivalent spatial coverage. If the coverage was not 100% the rates would not match because sensitive areas would not get targeted on one design but in another design they would be targeted. Using the test circuit as an example if only 50% of the circuit was targeted most of the flops sensitive to faults would be excluded from the test for the non-redundant structure, for the DWC structure only the primary copy of the interpolator would be covered. This results in the DWC structure having a greater possibility to have a sensitive flop targeted. If the coverage was not total in this way the DWC structure would produce more warnings than the non-redundant structure would produce errors given the same injection rates. A similar conclusion can be made for the TMR structure.

An additional limitation to the design is after ranging the pseudo-random flop selection value the distribution of ranged values is not even. Currently this distribution is biased towards the center of the distributions range. This distribution could be modified to shift the bias to any section of the distribution. The way this shift would be implemented is by subtracting a configurable value from the out-of-range values. As long as the subtracted value brings all the out of range values into the proper range the bias can be shifted without issue. This shift could be used to target more injections into fault sensitive areas of the circuit while still maintaining full coverage of the rest of the circuit. For the purposes of testing and regulation, an argument

could be made that injecting more errors into a part of the circuit known to be more sensitive to faults is a valid test.

Also some future work could be investigation into ways to alleviate some of the problems associated with the random number generators (RNGs). One option is the investigation of a RNG that allows for the specification of a range then provides a result that is in that range. Currently the RNGs algorithms that allow for range specification are based on rejection and regeneration. This means that they do not guarantee a properly ranged value within one clock cycle of the circuit.

An alternative solution to a method that generates a value within a specified range is a more efficient ranging method. The method of binning mentioned in section *3.13 Unique Design Methodologies and Choices* is a good method but is not resource efficient when implemented as a single clock cycle solution. If a design to perform proper ranging while being resource efficient and guaranteeing a valid result within a single clock cycle were to be used it would benefit the tool.

Another area of the design that could benefit from optimization is the design of the multi-hot decoder described in section *3.7 Addressing the Flip-Flops*. For the purposes of this thesis the design is acceptable but could benefit from optimization if possible. For example using the current design a module being tested with $2^k - 1$ flops supporting m injections requires m RGNs each generating a values between 1 and $2^k - 1$. Currently the multi-hot generation process decodes each k-bit value separately and then ORs each of the results together to generate the multi-hot code. If only n number of faults are being injected into the module during the test $m - n$ of the RNGs are disabled. An alternative design could only use one RNG and the resulting multi-hot decoder would only have a single hot output for input values of 1 to $2^k - 1$

however for input values greater than 2^k it could have up to m outputs hot. This would require a decoder that would decode a $log_2\lceil(2^k-1)^m\rceil$-bit value directly to a multi-hot code.

Although there are some limitations to the tool and some optimizations that could be made the tool is still very usable. It is possible to perform a variety of tests on a circuit and effectively analyses the results. This is particularly the case if the tester has a good understanding of the circuit being tested and of how the flop selection is distributed, allowing for effective tests to be preformed and analyzed.

Bibliography

Alfke, P. (1996). Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators.

Altera Corporation (2007). Stratix II device handbook.

Altera Corporation (2012). Cyclone III device handbook.

Asadi, G., Miremadi, S.-G., Zarandi, H.-R., and Ejlali, A. (2003). Fault injection into SRAM-based FPGAs for the analysis of SEU effects. In *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, pages 428–430.

Battezzati, N., Sterpone, L., and Violante, M. (2010). *Reconfigurable Field Programmable Gate Arrays for Mission-Critical Applications*. Springer.

Berrojo, L., Gonzalez, I., Corno, F., Reorda, M., Squillero, G., Entrena, L., and Lopez, C. (2002). New techniques for speeding-up fault-injection campaigns. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 847–852.

Carreira, J., Costa, D., and Silva, J. (1999). Fault injection spot-checks computer system dependability. *Spectrum, IEEE*, **36**(8), 50–55.

Cieslewski, G., George, A. D., and Jacobs, A. (2010). Acceleration of FPGA Fault Injection Through Multi-Bit Testing. In T. P. Plaks, D. Andrews, R. F. DeMara, H. Lam, J. Lee, C. Plessl, and G. Stitt, editors, *ERSA*, pages 218–224. CSREA Press.

Civera, P., Macchiarulo, L., Rebaudengo, M., Reorda, M. S., and Violante, M. (2002). An FPGA-based approach for speeding-up fault injection campaigns on safety-critical circuits. *Journal of Electronic Testing*, **18**(3), 261–271.

Fabula J., Moore J., W. A. (2007). Understanding neutron single-event phenomena in FPGAs. *Military Embedded Systems*.

Knight, J. C. (2002). Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550, New York, NY, USA. ACM.

Lawford, M., Bergstra, J. P., Deng, H., Eles, C., Sullivan, J., and Trachimowich, J. (2010). Report 3: Feasibility Study of FPGA Based Platforms for Safety-Critical Systems. Technical Report 3, McMaster University McSCert.

Lesea, A., Drimer, S., Fabula, J., Carmichael, C., and Alfke, P. (2005). The rosetta experiment: atmospheric soft error rate testing in differing technology FPGAs. *Device and Materials Reliability, IEEE Transactions on*, **5**(3), 317–328.

Lima, F., Carmichael, C., Fabula, J., Padovani, R., and Reis, R. (2001). A fault injection analysis of Virtex FPGA TMR design methodology. In *Radiation and Its Effects on Components and Systems, 2001. 6th European Conference on*, pages 275–282.

Montgomery, D. and Runger, G. (2007). *Applied statistics and probability for engineers*. Wiley.

Nazar, G. and Carro, L. (2012). Fast single-FPGA fault injection platform. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2012 IEEE International Symposium on*, pages 152–157.

Rajski, J., Tamarapalli, N., and Tyszer, J. (2000). Automated synthesis of phase shifters for built-in self-test applications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **19**(10), 1175–1188.

Sauer, M., Tomashevich, V., Muller, J., Lewis, M., Spilla, A., Polian, I., Becker, B., and Burgard, W. (2011). An FPGA-based framework for run-time injection and analysis of soft errors in microprocessors. In *On-Line Testing Symposium (IOLTS), 2011 IEEE 17th International*, pages 182–185. IEEE.

Spilla, A., Polian, I., Müller, J., Lewis, M., Tomashevich, V., Becker, B., and Burgard, W. (2011). Run-time Soft Error Injection and Testing of a Microprocessor using FPGAs.

Ubar, R., Raik, J., and Vierhaus, H. T., editors (c2011). *Design and test technology for dependable systems-on-chip*. IGI Global (701 E. Chocolate Avenue Hershey Pennsylvania 17033 USA), Hershey, Pa.

Velazco, R., Fouillat, P., and Reis, R. (2007). *Radiation effects on embedded systems*. Springer, Dordrecht.