2015-12-01

Configuration Scrubbing Architectures for High-Reliability FPGA Systems

Aaron Gerald Stoddard
Brigham Young University

Configuration Scrubbing Architectures for High-Reliability FPGA Systems

Aaron Gerald Stoddard

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

Master of Science

Michael J. Wirthlin, Chair
Brent E. Nelson
Gregory P. Nordin

Department of Electrical and Computer Engineering

Brigham Young University

December 2015

ABSTRACT

Configuration Scrubbing Architectures for High-Reliability FPGA Systems

Aaron Gerald Stoddard
Department of Electrical and Computer Engineering, BYU
Master of Science

Field Programmable Gate Arrays (FPGAs) are being used more frequently in space applications because of their reconfigurability and intensive processing capabilities. FPGAs in environments like space are susceptible to ionizing radiation which can cause Single Event Upsets (SEUs) in the FPGA's configuration memory. These upsets may cause the programmed user design on the FPGA to deviate from its normal behavior. Space missions cannot afford to allow important data processing applications to become corrupted due to these radiation upsets.

Configuration scrubbing is an upset mitigation technique that detects and corrects upsets in an FPGA's configuration memory. Configuration scrubbing periodically monitors an FPGA's configuration memory utilizing mechanisms such as Error Correction Codes (ECCs), Cyclic Redundancy Checks (CRCs), a protected golden file, and partial reconfiguration to detect and correct upset memory bits. This work presents improved Xilinx 7-Series configuration scrubbing architectures that achieve minimal hardware footprints, competitive performance metrics, and robust detection and correction capabilities.

The two principal scrubbing architectures presented in this work are the readback and hybrid scrubbers which detect and correct Single Bit Upsets (SBUs) and Multi-Bit Upsets (MBUs). Harnessing the performance advantages granted by the 7-Series internal Readback CRC scan, a hybrid scrubber built in software for the Zynq XZC07020 FPGA has been measured to correct SBUs in 8.024 ms, even-numbered MBUs in 13.38 ms, and odd-numbered MBUs in 21.40 ms. It can also perform a full readback scrub of the entire device in under two seconds. These scrubbing architectures were validated in radiation beam tests, where one of the architectures corrected MBUs as large as sixteen bits in a single frame.

Keywords: FPGA, radiation testing, BYU, scrubbing, configuration, readback, reliability, TMR, SEU, ECC, CRC, MBU, Zynq, PCAP, Xilinx, bitstream, upset mitigation, 7-Series

ACKNOWLEDGMENTS

This work could not have been accomplished without the help of my advisor, Dr. Michael Wirthlin. His encouragement and tutelage were paramount to overcoming each obstacle that we encountered. Dr. Brent Nelson and Dr. Gregory Nordin were also instrumental in improving the clarity and cohesion of my writing.

I sincerely appreciate the members of the scrubbing team in the BYU Configurable Computing Lab. Without their help, this work would not be possible. Ammon Gruwell, Alex Harding, and Peter Zabriskie were all vital in developing innovative ideas to tackle the many complications that we faced in our scrubbing journey. I would also like to thank Alex Wilson and Jordan Anderson, who integrated the Readback and Hybrid Scrubbers into the Linux CSP flight software, making it possible for these scrubbers to actually go up into outer space. They also made it possible to get results from these scrubbers functioning in radiation beam tests.

My gratitude also goes to CISCO and to the Los Alamos Neutron Science Center (LANSCE), which provided valuable radiation beam time to validate the scrubbers presented in this thesis. Their technical staffs and facilities were tremendously helpful.

Finally, I could not have done what I did without the continuing support of my family, particularly from my uncle, who first inspired me to pursue my education in this field; my father, who provided significant logistical assistance to make my education possible; and my wife, whose daily support, patience, and constant encouragement has been an unending source of motivation and determination.

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

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As space exploration continues to be a priority of both industrial and governmental endeavors, improved technologies are needed to ensure the reliability of high-performance processing applications in high-radiation environments. Present satellite missions require more intensive real-time computations that must have high reliability and operational performance for on-board image and video processing sensor units [5].

The Solar Orbiter is one such satellite whose mission is to observe the sun at close proximity [6]. An on-board telescope records high-resolution 2048 x 2048 images of the sun at six different wavelengths and four polarization states. The satellite must process a total of 6.44 Gbits/min to acquire important magnetic field data [7]. Unfortunately, current generations of space-qualified processors cannot perform this computation in the alloted time frame. Instead, a different type of technology is needed.

Field Programmable Gate Arrays (FPGAs), are becoming an increasingly popular alternative to space-qualified processors [6], [8]. FPGAs provide near equivalent processing capabilities for space applications, customizable circuit design, and fast reprogrammability compared to the conventionally used radiation-hardened processors [9]. As such, FPGAs have become a realistic candidate for use in space-mission applications. The Solar Orbiter, along with space missions such as the STP-H5/ISEM and CeREs [4], have been equipped with FPGAs to handle their data processing needs.

One significant drawback to using SRAM-based FPGAs is their susceptibility to ionizing radiation. In a space environment, ionizing radiation can come from a variety of sources including solar flares, harsh space weather events, and cosmic rays [10]. When this radiation comes into contact with the SRAM memory cells of an FPGA, the data held in those cells can become corrupted. Since an FPGA's configuration memory stores the implementation

of a user design, radiation upsets which change bits in the FPGA's configuration memory can cause the design to function improperly.

One way of protecting against radiation-induced upsets has been achieved by fabricating the FPGA to be Radiation-Hardened by Design (RHBD) or by using Radiation-Hardened by Process (RHBP) techniques [11], [12]. Such fabrication protects the configuration cells at the silicon and transistor levels either by doping the silicon differently (RHBP), or by using more transistors for a single cell with wider spacing between them (RHBD). Unfortunately, radiation-hardened FPGAs are substantially more expensive [13] and do not exist for more recent generations of FPGA device families, which are significantly larger and have higher processing capabilities than their predecessors [14].

An alternative to using RHBD FPGAs is to implement one or more upset mitigation techniques to monitor and correct upsets in the FPGA's configuration memory. An effective upset mitigation approach to improve reliability is to combine Triple Modular Redundancy (TMR) with configuration scrubbing [15]. TMR triplicates a user circuit in the configuration memory and uses majority voters to tolerate upsets in one of the three modules while still producing a correct output with the other two. Scrubbing is the repair mechanism that corrects the upsets of the affected module before the other two become corrupted as well.

Several FPGA configuration scrubbing architectures have been previously developed using a variety of strategies, interfaces, and specifications [9]. These scrubbing architectures were built primarily for earlier Xilinx FPGA families such as the Virtex-4, Virtex-5, and Virtex-6 families. To the author's knowledge, only two known architectures have been developed for the 7-Series devices [3], [16]. The purpose of this thesis is to present novel Xilinx 7-Series scrubbing architectures that can be incorporated into current and future FPGA designs to help improve overall reliability. The scrubbing architectures in this thesis are targeted for protecting FPGAs in high-radiation environments such as outer space.

One of the important contributions of this work is a new hybrid scrubbing architecture for the 7-Series FPGAs. Inspired by the two existing 7-Series scrubbers [3], [16], this hybrid scrubber utilizes the built-in internal configuration memory scanner known as the Readback CRC. This mechanism can be exploited to detect all configuration memory upsets via CRC and ECC encodings and to automatically correct Single Bit Upsets (SBUs). The Readback

CRC achieves high performance in upset detection and SBU correction. It also avoids the need for the user to create a complex internal scrubber circuit that would occupy valuable FPGA resources to implement. While the Readback CRC has inherent limitations such as not being able to correct MBUs, a more robust scrubber is used to correct larger upsets.

Furthermore, the release of the Zynq-7000 System-On-Chip (SoC) family provides a new avenue in the development of scrubbing architectures. This FPGA family features a dual-core ARM Cortex-A9 processing system (PS) alongside the FPGA's programmable logic (PL). Built into the PS is a new configuration interface known as the Processor Configuration Access Port (PCAP) which grants the processors access to the FPGA's PL. The processors can run software scrubbing code which uses the PCAP to monitor and correct upsets in the PL. Any scrubbing logic that previously had to be implemented using FPGA resources can now be converted into software code, enabling faster compilation and debugging of the scrubbing architecture. The scrubbers in this thesis were validated on Zynq SoCs.

The FPGAs considered in this thesis are limited to SRAM-based FPGAs as opposed to anti-fuse, flash-based, or other types of FPGAs. The programming cells of anti-fuse FPGAs are not sensitive to radiation, but they also do not support the ability to be reconfigured in the field [17]. Flash-based FPGAs are generally smaller and less flexible than SRAM-based FPGAs [9]. Furthermore, the FPGAs considered in this work are exclusively Xilinx FPGAs.

1.1 Thesis Contributions

The primary contributions of this thesis are as follows:

1. A comprehensive reference of scrubbing configuration commands, operations, and hardware necessary for building 7-Series scrubbers.

2. The presentation of three scrubbing architectures for the 7-Series devices. One of these architectures is a novel hybrid scrubbing architecture.

3. Validated software implementations of the 7-Series scrubbing architectures for the Zynq-7000 SoC family, including radiation beam test setup and results. These results include the number of corrupted bits in a single upset and the number of occurrences of that upset type.

These scrubbers are ideal for space applications and high-energy physics experiments which must function reliably in environments with high upset rates. FPGA users can customize these transparent scrubbing architectures developed by this thesis to meet the needs of their digital designs.

1.2 Thesis Organization

Chapter 2 provides a general background of ionizing radiation. This background discusses where ionizing radiation comes from, presents the effects of ionizing radiation on FPGAs, and explains how each new generation of FPGAs is increasingly more susceptible to ionizing radiation than the previous generation. Chapter 2 will also introduce common upset mitigation mechanisms that have been implemented previously to handle radiation upsets. Variations of these mechanisms serve as building blocks for the architectures presented in this thesis.

Chapter 3 explains the configuration details necessary for implementing scrubbers on Xilinx FPGAs. The basic layout of FPGAs, bitstreams, and configuration operations are all described in this chapter. Chapter 4 goes a step further to introduce the built-in SEU Configuration Memory Protection components for 7-Series devices, the most notable of which is the Readback CRC internal scan. The Readback CRC scan is heavily relied on by the hybrid scrubbing architecture described Chapter 6. The scan's shortcomings are also discussed, including the types of upsets that the scan fails to correct. Chapter 5 reviews existing scrubbing solutions developed primarily for older FPGA families. This chapter also gives an overview of the basic scrubbing strategies employed by many scrubbing architectures.

Chapter 6 presents the 7-Series scrubbing architectures developed by this thesis with complete methodologies and explanations of the files needed to implement the architectures. Examples of two of these architectures implemented on the Zynq FPGA family are given in Chapter 7. Chapter 7 also gives the results of radiation tests performed on Zynq FPGAs running the readback and hybrid scrubbing architectures. The performance metrics of these architectures is also described in this chapter. This work will conclude in Chapter 8.

Chapter 2

FPGA Reliability in High-Radiation Environments

SRAM-based FPGAs are sensitive to ionizing radiation [10]. As FPGA transistor size decreases, radiation particle strikes have a greater impact on the SRAM memory cells of FPGAs. When used in space or other radiation environments, SRAM-based FPGAs require a certain level of reliability and protection from radiation to function properly.

This chapter will address how upsets occur from ionizing radiation, where the radiation comes from, and how an FPGA can be detrimentally affected. In addition, this chapter will give a succinct background of mitigation techniques employed to remedy these upsets. Understanding the effects of radiation upsets as well as the foundational mitigation strategies employed to handle them is important to comprehending the configuration scrubbing techniques discussed later in this thesis.

2.1 Ionizing Radiation Overview

Ionizing radiation is any type of particle or electromagnetic wave that carries enough energy to ionize or remove electrons from an atom [18]. For transistors that make up memory cells (such as SRAM cells), ionizing radiation can disrupt the transistor state by causing surges of current to flow or by stopping currents from flowing. Ionizing radiation comes from a variety of sources and has a wide range of effects on FPGAs. These effects are considered in this work in the context of both space and high-energy physics experiments.

2.1.1 Radiation Sources

The most common forms of ionizing radiation particles are high-energy neutrons, heavy ions, and protons. FPGAs in space are susceptible to several types of ionizing radiation sources. Cosmic rays from galactic origins, which often contain high-energy neutrons, occur

frequently outside the Earth's atmosphere. Unplanned space weather events such as solar flares and coronal mass ejections are also potential radiation sources [10]. Other sources of ionizing radiation in outer space include solar particles with protons and heavy ions, as well as particles trapped in Earth's magnetic field.

Radiation can affect electronics on Earth through means such as alpha-particles and neutron-induced Boron Fission [19]. Uranium and thorium impurities inside the physical semiconductor connections of a circuit chip can sometimes decay to lower energy states and, while so doing, emit alpha particles. Alpha particles are composed of two neutrons and two protons, and, once emitted, can ionize silicon atoms of electronic devices. Moreover, if cosmic rays consisting of thermal neutrons bombard an isotope of Boron (10B), an element commonly used as a p-type dopant in silicon, both gamma rays and alpha particles will be emitted [20].

FPGAs are also being used in high-energy physics experiments at several facilities around the world [10]. These experiments use particle accelerators to cause charged particles to travel at high speeds and then collide. The resulting collisions create a number of byproducts that are studied to better understand subatomic particle interactions. FPGAs are often used in these experiments to carry out tasks such as interfacing sensors (analog to digital converters), measuring nanosecond time differences, and streaming data outside of the experiment via serial I/O. Naturally, FPGAs are very susceptible to the resulting radiation fields generated by these particle collisions. Luckily, the flux of radiation particles is typically constant and thus predictable compared to a space environment. In addition, the experiment can always be shut down, and the FPGA can be examined if it malfunctions.

2.1.2 Radiation: Single Event Effects

When radiation particles collide with the semiconductor lattice of transistors, their energy is transferred to the atomic particles of the transistor. These radiation strikes may cause depletion regions to form in the transistors by leaving behind an excess of electrons or holes in their wake [21]. Depending on the orientation of the free electrons or holes generated, electrical current may begin to flow, or cease flowing, which in turn could cause the digital

circuit to change its logical state (see Figure 2.1). When such an event occurs in an FPGA, it is known as a Single Event Upset (SEU).

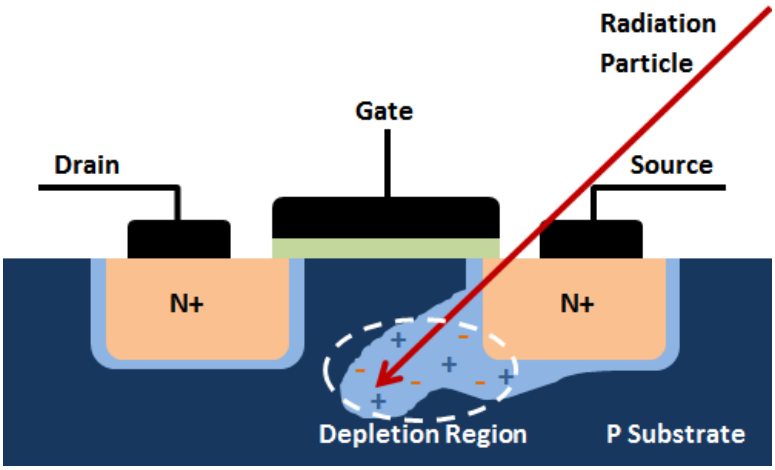

Figure 2.1: The Effect of a Radiation Particle Strike on a Transistor

Xilinx FPGA configuration memory is organized into units called frames which are the smallest addressable units on an FPGA. They are typically made up of 40 to 120 32-bit words each (depending on the FPGA family). These frames are the memories that scrubbers monitor. If an SEU only affects one logical bit in a frame, it is known as a Single-Bit Upset (SBU). SBUs in different frames are each classified as separate SBUs for correction purposes.

When multiple upsets occur in the same frame, the result is a Multi-Bit Upset (MBU). A single event that causes multiple bits in the same frame to upset is a Multi-Cell Upset (MCU) [22]. This thesis classifies MCUs as MBUs for simplification purposes. As noted in [23], MBUs are increasingly more likely as the transistor process technology continues to shrink. The same-size atomic particles bombard ever smaller device geometries, causing increasing numbers of upset events.

An FPGA's configuration memory defines the digital circuit that the FPGA implements. The configuration memory is made up of configurable resources such as Look-Up-Tables (LUTs), Flip-Flops (FFs), and other state elements. Furthermore, the routing logic that connects these components is also a part of the configuration memory.

Upsets to SRAM configuration cells can actually change the functionality of the circuit design. An example of a digital circuit implemented on an FPGA is shown in Figure 2.2. The logical bits of the configuration memory are shown defining the routing and function of FPGA resources used to implement a two-input AND gate.

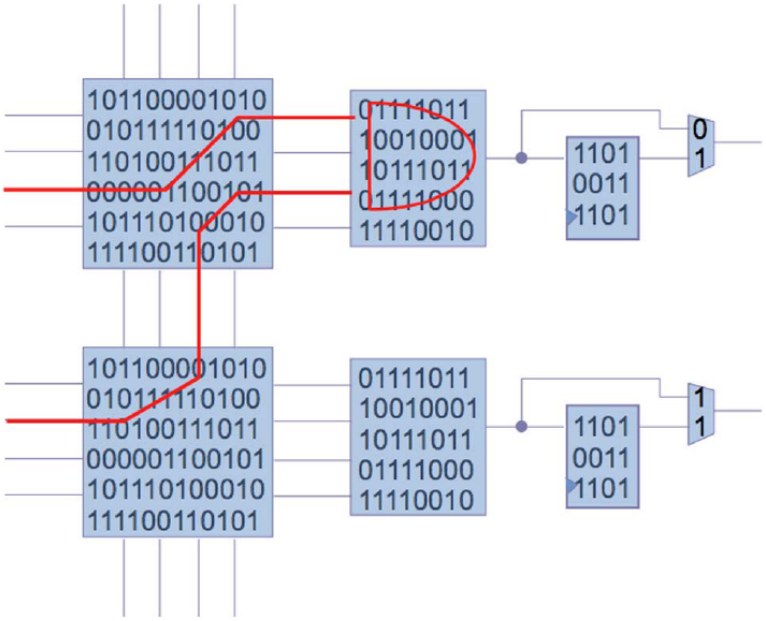

Figure 2.2: Routing and Logic Before Upset

When one upset occurs in the routing logic, as shown in Figure 2.3, one of the inputs to the gate is disconnected. When another upset occurs inside one of the LUTs, the AND gate is changed to an XOR gate. The exact function of the original circuit has been lost.

It is important to note that not all upsets to configuration bits cause problems. Many digital circuit designs do not fully utilize all of the resources on an FPGA. Upsets to these areas will not necessarily cause critical failures to the user design. Furthermore, radiation upsets are typically not permanent in the sense that no lasting damage to FPGA transistors occurs.

Configuration bits which cause digital circuits to deviate from their normal functionality are known as sensitive bits [10]. If these bits are upset, the circuit will cease to perform reliably. A significant amount of research has been done measuring how many sensitive bits

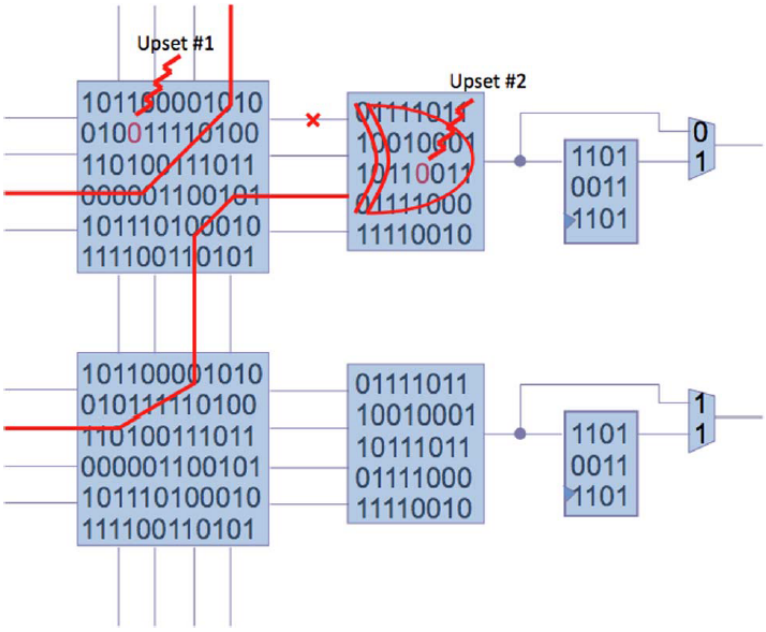

Figure 2.3: Routing and Logic After Upset

various designs contain to give users an idea of how fragile their designs are, or how likely the designs are to be upset [24], [25].

2.1.3 FPGA Transistor Density

With each FPGA family that Xilinx produces, the transistor size has dramatically decreased, enabling more configuration logic cells to be packed into the same amount of physical area [26]. This trend is due, in part, to the smaller process technology of the transistors of each family. The amount of area affected by one radiation particle strike has not changed, but the number of logic cells and transistors inside that area has increased. The smaller transistor size also decreases the energy necessary for the transistor threshold voltage to be met or surpassed, thus making it easier for the logical value stored in the SRAM cell to upset [21]. Table 2.1 shows the number of configuration bits of the largest and smallest of each of the more recent Xilinx FPGA families [27], [28], [29], [1], [30], [31].

The Failures in Time (FIT) rate indicates the number of failures a certain area receives every billion (10^9) device operation hours. The FIT rates, shown in Table 2.4, are determined from real time measurements from radiation beam testing and are given as FIT/Mb, (Mb=

Table 2.1: Largest and Smallest Bitstream Sizes for Recent Xilinx Families

Family	nm	Device	Min Bits	Device	Max Bits	FIT/Mb
Virtex-4	90	XC4VLX15	4,765,184	XC4VLX200	51,367,424	263
Virtex-5	65	XC5VLX20T	6,251,200	XC5VLX330T	82,696,192	165
Virtex-6	40	XC6VLX75T	26,239,328	XC6VHX565T	160,655,264	105
7-Series	28	XC7VX330T	111,238,240	XC7V2000T	447,337,216	85
UltraScale	20	XCVU065	200,713,824	XCVU440	1,031,730,976	38

10^6 bits). What is counterintuitive about these numbers is that the FIT/Megabit (Mb) is *decreasing* with each newer FPGA family, suggesting that these newer families actually incur fewer failures. However, because each family is also increasing significantly in configuration bit amounts (due to smaller process technologies), the FIT/Device is actually increasing overall.

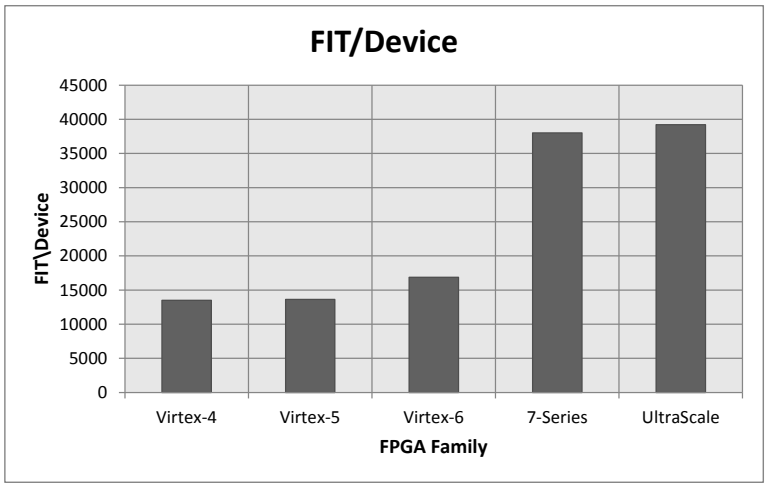

Figure 2.4: FIT/Device for Xilinx Families [1]

Figure 2.4 shows the FIT/Device that takes into account both the FIT/Mb and the number of Mb for each Xilinx FPGA family. The trend is clear that the since each Mb of memory is smaller (due to transistor size), a larger amount of Mb physically on the device is

[1]These measurements were computed using the FIT/Mb for CRAM only. A more accurate result could be obtained by calculating the number of CRAM and BRAM bits separately and multiplying each of those with their corresponding FIT/Mb and adding the two results together.

more susceptible to failures despite a decreasing FIT/Mb. The reality is that the FIT/Device is increasing. Hence, scrubbing these upsets becomes all the more important with each new FPGA family.

2.2 General FPGA Upset Mitigation Techniques

Several basic error mitigation techniques have been developed to improve FPGA reliability. Many current FPGA reliability solutions combine several of these mitigation techniques. A summary of the most common techniques is given below; however, this summary is not meant to be exhaustive. These upset mitigation techniques are used in several high-speed data transmission and memory systems all across the electronics spectrum. The summaries presented here will describe how some upset mitigation techniques are implemented specifically with regard to Xilinx FPGAs. Configuration scrubbing is only one of the many upset mitigation options available, but it is one of the more important options.

2.2.1 Radiation Hardening

Radiation hardening is commonly achieved by using one of two methods. Radiation-hardened by design (RHBD) is a redundancy process where more transistors are used to build one SRAM cell. The transistors are placed in a special layout such that the probability that the same ion collides with multiple transistors of the same SRAM cell is very low, making it improbable to cause an upset [32]. The Virtex-5QV (released July 2010) is the most recent Xilinx RHBD FPGA despite the release of three subsequent Xilinx families [12].

Radiation-hardened by process (RHBP) is where the FPGA is fabricated in a way that the transistors are protected from ionizing radiation at the silicon level [11]. Gated resistor hardening is one example of RHBP that uses a variable resistor to increase the threshold voltage required to change the state of the memory cell. Another method is to preserve circuit performance by lowering the threshold voltage when the write enable to the memory cell goes high [32].

Using radiation-hardened parts is a safe option, but the parts are typically much more expensive than commercial off-the-shelf (COTS) parts. If near-perfect reliability is required, then this technique is probably the best option. However, if reliability requirements are

more flexible, then the other mechanisms presented in this work are much more cost-effective alternatives.

2.2.2 Error Correction Code

An Error Correction Code (ECC) is a redundancy coding mechanism that is useful for correcting SBUs. ECCs achieve Single-Error Correction and Double-Error Detection (SECDED) on a per-frame basis. Each configuration frame in Xilinx FPGAs contains an ECC word to provide basic SBU correction. Double-bit upsets are detected when using an ECC but are not able to be corrected.

An ECC is implemented in hardware with a dedicated circuit consisting of XOR chains that combine specific bits in a frame to compute a checksum, known as a syndrome[2]. Every bit location in the frame is accounted for somewhere in the circuit [33]

The ECC by itself does not actually do any upset mitigation. A controller circuit is needed to decode the ECC and run the frame data through the ECC circuit. The result indicates where in the frame an SBU (if any) is located, or if a double-bit upset has occurred. The controller can then use that information to correct the upset(s). A syndrome of all zeros indicates that the ECC was satisfied. A "satisfied" ECC usually means that there are no errors. However, it is quite possible that enough bits in the right places are upset in the frame so as to not be detected by the ECC. The ECC word inside the frame, known as the "golden ECC," is generated to be the exact value that bridges the gap between the bits in the frame and a zero (satisfied) syndrome.

ECCs in Xilinx FPGAs detect all odd-numbered upsets [3]. They also detect all two-bit upsets. Even-numbered upsets higher than two may go undetected because a sufficient number of bits are upset in just the right places so as to not be caught by the ECC, but such an instance is quite rare. Section 4.1.2 contains more detailed information about the ECCs on Xilinx FPGAs.

[2]Refer to the file FRAME_ECCE2.vhd v.11.1 (included in Xilinx Vivado software tools distributions) for the ECC circuit description for Xilinx 7-Series devices.

2.2.3 Cyclic Redundancy Check

A Cyclic Redundancy Check (CRC) is another redundant coding mechanism that is useful for upset detection. Its primary advantage compared to other coding mechanisms like ECCs is its ability to detect upsets within millions of bits using a single CRC value. Because of the heavy redundancy, changing even one bit of the configuration will result in a vastly different CRC value. Its main disadvantage, however, is that it cannot identify where the errors are specifically located. Thus, it is used primarily as an error detection mechanism.

A circuit used to compute a CRC is based off of the remainder of a carefully designed polynomial division circuit. CRCs are added to the configuration memory of FPGAs to provide a means of checking data integrity. One 32-bit value is calculated for the entire configuration memory. Each frame of an FPGA's configuration contributes to this calculation, yet the CRC does not have to expand in bit-length with each subsequent frame (hence the "redundancy").

CRCs are particularly useful in detecting MBUs across an FPGA. Such a feature is invaluable when other codes like ECCs can only detect odd-numbered and specific even-numbered upsets (see Section 2.2.2). According to [34], the 32-bit CRC on Xilinx devices detects any 31 bit error in a single frame with 100 percent accuracy. Beyond 31 bits, coverage is still very good, but a few cases exist where the MBU would go undetected. With such high error coverage, the probability that a configuration upset goes undetected is very low.

CRCs play a significant role in several scrubbing architectures, including the hybrid scrubbing architecture presented by this thesis. Chapter 4 gives more details about the CRCs in Xilinx FPGAs, while Chapter 6 illustrates how the scrubbers of this thesis use the CRCs in their architectures.

2.2.4 Triple Modular Redundancy

Triple Modular Redundancy (TMR) is an invaluable upset mitigation technique for improving FPGA reliability. Unlike the techniques presented previously which were built into the hardware of the FPGA, TMR is a technique that designers must incorporate into their circuit. In a design with TMR, the design circuitry is triplicated into three identical copies and typically placed at three different locations in the FPGA. The outputs (and

sometimes the inputs) of all of these circuits are connected to a common voter circuit which compares the individual circuit outputs and chooses the majority value. To avoid a single point of failure in the voter circuit, the voters are often triplicated as well (see Figure 2.5).

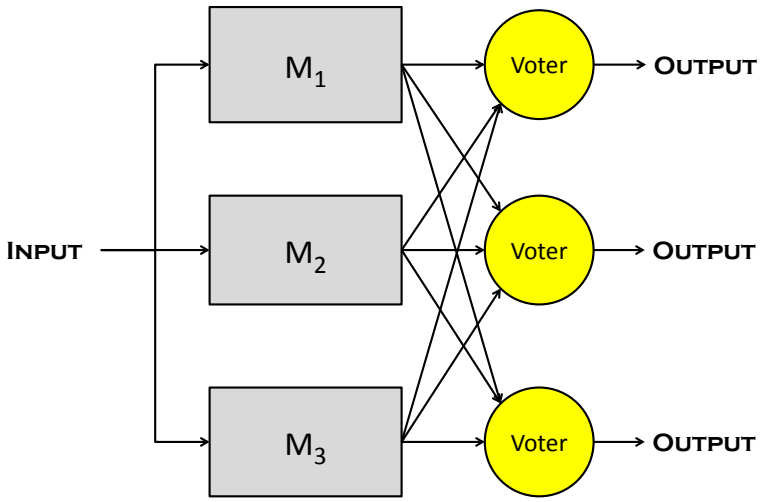

Figure 2.5: TMR Circuit

If an upset occurs in one of the three modules, then the other two modules will continue to produce a correct result overall by out-voting the faulty module. However, once a second module receives an upset, then TMR fails.

TMR, by itself, does not actually have an improved reliability for longer mission times compared to just using a single module alone [35]. When used with a repair mechanism, TMR reliability dramatically improves [36]. Appendix A gives the standard Markov equations for TMR analysis. Clearly, TMR with repair has a significantly higher mean time to failure (MTTF), and an improved reliability curve, especially when the repair rate μ is much greater than λ, the upset rate. However, TMR does come with at least three times the area and power costs compared to non-TMR systems [37].

When an upset occurs in one of the TMR modules, the error needs to be repaired quickly before another upset affects a second module, causing TMR to fail. TMR prolongs the reliability of the circuit granting the repair mechanism time to fix the first error before

another module is upset and the overall circuit fails. The focus of this thesis, configuration scrubbing, is the repair mechanism often used in FPGA-based TMR systems [9], [15]. Working in tandem, TMR and scrubbing can achieve incredible reliability improvements [36].

It is worth noting that just having a repair mechanism (like scrubbing) without TMR would fail instantly if the ionizing radiation were to upset a sensitive bit and cause the circuit to malfunction. Thus, both TMR and scrubbing are necessary to achieve an overall reliability improvement.

2.2.5 Memory Scrubbing

Unlike the static configuration memory of an FPGA, dynamic memories change constantly with reads and writes by an application. Dynamic memories also incur upsets that need to be mitigated, but the methods used are somewhat different than those used for static memories. Appendix E gives further discussion on mitigating upsets in dynamic memories.

Memory scrubbing can be performed on memories such as the Block RAMs (BRAMs) of an FPGA. These scrubbers will often use variations of ECCs and/or TMR [37]. For instance, each block of memory may contain an ECC word, and a dedicated memory bank is used to store the golden ECC values. Whenever the block of memory is modified, a new ECC is computed and stored in the memory bank. Another mechanism can then periodically compare the ECC values in each memory to the corresponding value in the bank and correct SBUs if mismatches occur.

TMR could be used by triplicating the memory and reading from each memory periodically to see if a difference in the memories exists. TMR can then use the value agreed upon by the majority vote to then correct the upset in the location that differs [38].

The actual scrubbing of the memory is performed either probabilistically or deterministically [39]. Probabilistic scrubbing is when the blocks of memory are only scrubbed whenever that memory is actually accessed (via a read or write). However, if portions of memory are never accessed, then upsets could build up in those memories. Deterministic scrubbing is a continuous sequential scan that always checks all memories. Deterministic

scrubbing is naturally more power-consuming and performance-taxing, yet achieves a better reliability compared to probabilistic scrubbing.

2.2.6 Configuration Scrubbing

Configuration memory is expected to remain static throughout the life of the digital design. Configuration scrubbing involves periodically checking this memory, detecting upsets, then using dynamic partial reconfiguration to overwrite the upsets [17]. The initial configuration must first be complete before any scrubbing can occur. Configuration scrubbing is usually performed at periodic intervals while the FPGA is in operation (hence "dynamic" reconfiguration). This partial reconfiguration is not intended to interrupt normal FPGA operation.

Configuration scrubbing has two primary responsibilities: error detection and correction. A scrubber is a controller that determines when and where to perform a partial reconfiguration for correction; scrubbers also periodically check the memory to detect upsets. Scrubbers come with a variety of trade-offs, including size, power consumption, performance, robustness, etc. Chapter 5 presents several examples of configuration scrubbers using various detection and correction mechanisms.

Recognizing the real threat that radiation poses to FPGAs in high-radiation environments motivates the need for improved upset mitigation techniques. Several basic upset mitigation techniques were highlighted in this chapter that are used in the scrubbing architectures presented in this thesis. The focus of this work is configuration scrubbing upset mitigation mechanisms. To understand how configuration scrubbing is performed, one must first understand the configuration mechanisms of an FPGA.

Chapter 3

Configuration of Xilinx FPGAs

Configuration scrubbing is based on using the built-in mechanisms that configure an FPGA device. It is necessary to first understand these configuration mechanisms in order to understand how scrubbing is performed. While all of the information described in this chapter is documented in official user guides, this chapter will clarify topics that might not be easily understood by an initial reading of the documentation. References to the original documentation will also be included as appropriate throughout the chapter.

While some low-level details of FPGA configuration differ between Xilinx FPGA families, the underlying process is the same. The purpose of this chapter is to present a general reference of the essential configuration components and processes used in configuring an FPGA, particularly about those related to scrubbing. Important configuration details include a summary of Xilinx FPGA architecture, an explanation of configuration bitstreams, the commands used to both read and write to the configuration memory, and other configuration nuances and procedures that a scrubber must account for. Most of the information presented in this chapter is explicitly given in the 7-Series Configuration Guide [1].

3.1 Xilinx FPGA Architecture

The standard configurable resource in a Xilinx FPGA is known as a Configuration Logic Block (CLB). CLBs are the most prevalent of the physical building blocks that are connected together to make up a user's design in hardware. CLBs are tiled across an FPGA in rows and columns with channels of routing wires lying in between them (see Figure 3.1). CLBs contain various amounts of logic resources. For instance, a 7-Series CLB contains eight 6-input Look-up-Tables (LUTs), 16 Flip-Flops (FFs), arithmetic carry logic, and wide-function multiplexers [40]. Some CLBs contain elements that support the ability to im-

plement shift registers and distributed RAM. Table 3.1 gives a breakdown of CLB resources for the smallest and largest 7-Series Virtex FPGAs.

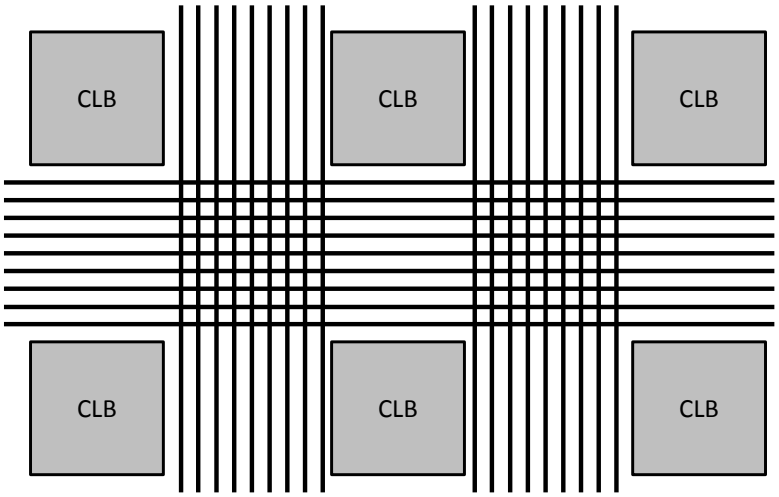

Figure 3.1: Tiled Configuration Logic Block Layout

Table 3.1: CLB Resources for Virtex 7-Series FPGAs

Device	CLBs	6-input LUTs	Dist RAM	Shift Reg	Flip-Flops
7VX330T	25,500	204,000	4,388 (KB)	2,194 (KB)	408,000
7V2000T	152,700	1,221,600	21,550 (KB)	10,775 (KB)	2,443,200

3.2 Configuration Bitstreams

CLBs are enabled and programmed with bits from a unique binary file generated by Xilinx software tools. This file is called a "bitstream." The bitstream contains all of the information necessary to program the specific elements inside each CLB that will be a part of the user design. This information includes which LUTs and FFs should be enabled, what the LUT equations are, which routing PIPs (Programmable Interconnect Points) should be turned on to wire the design components together, etc. Each of these mechanisms is

controlled by one or more bits in a bitstream. When users wish to modify their design, they simply generate a new bitstream to program to the FPGA.

3.2.1 Frames

Bitstreams are divided into groups of 32-bit words known as frames. Frames are the smallest addressable unit of the configuration memory space [1]. All configuration operations must act on one or more frames, as opposed to individual words or bits. Table 3.2 gives the number of 32-bit words in a single frame for the most recent Xilinx device families.

Table 3.2: Number of 32-bit Words in a Frame for Different Xilinx Families

Xilinx Family	Words per Frame
Virtex-4	41
Virtex-5	41
Virtex-6	81
7-Series	101
UltraScale	123

The mapping between the bits in the bitstream and the specific FPGA resource that the bit controls is not made public by Xilinx. Fortunately, this information is not needed to implement configuration scrubbing.

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

Figure 3.2: Frame Address Register Description [1], p. 106

Each frame has a frame address (FRAD) that specifies its location on the FPGA. Figure 3.2 shows the breakdown of the address bits of the FRAD. One of the important segments of the FRAD is the block type. Determined by the high three bits of its address, each block type has a specific function dealing with different resource components of the FPGA. Based on Figure 3.2, typical bitstreams contain Block Types 0, and 1. In practice, bitstreams almost never contain Types 2, 3, or 4. Information about these types, if any, is not available. A more detailed explanation of the different frame types found in a 7-Series FPGA is given in Section 4.1.1.

3.2.2 Bitstream Composition

The bitstream is organized in a common pattern for every Xilinx FPGA. Bitstreams are organized as shown in Figure 3.3. Each of the elements in Figure 3.3 is defined below.

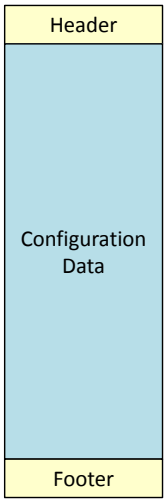

Figure 3.3: Bitstream Composition

1. **Header**

   This section contains readable ASCII characters, including a creation time stamp, followed by the initial configuration commands. These commands are a series of 32-bit words that write to the internal configuration registers (see Section 3.3.3) to prepare the FPGA to receive the subsequent configuration data. These commands include

initializing the CRC check, setting the configuration and control options, setting the frame address register (FAR) to the starting address, and issuing the write configuration command. Note that none of these commands are considered to be apart of frames. [1]

2. **Configuration Frame Data**

   As described in Section 3.2.1, configuration data is organized into frames which make up the bulk of a bitstream. It should be noted that the configuration data is further subdivided into configuration memory (Type 0) frames and Block RAM (Type 1) frames (see Section 4.1.1).

3. **Footer**

   After the configuration data, a final footer section is included which triggers a device-wide CRC check, initializes all flip-flops, begins the STARTUP sequence, and desynchronizes the FPGA configuration module from accepting further commands.

The exact size of a bitstream depends on the number of frames in the device, which depends on the number of logic resources (CLBs). Devices with more logic resources have more frames and, consequently, larger bitstreams. In essence, knowing the FPGA's bitstream size gives an indication as to the amount of logic resources available on that FPGA. Table 2.1 provides a comparison of bitstream sizes for different Xilinx FPGA families.

3.3 Configuration Module

The unit which processes the bitstream data and carries out the encoded instructions is called the configuration module in this thesis (its actual name is not officially known). This module is the singular gateway into and out of the configuration memory. The configuration module is responsible for evaluating bitstream packets and placing the bitstream data into the configuration memory. Inside the configuration module lie a series of dedicated hardware registers which control configuration modes and options. All configuration operations are performed by reading or writing to these registers [1].

[1] See Table 3.4 later in the chapter for the sequence of a sample bitstream.

3.3.1 Configuration Process

The configuration process begins by powering up the device, and clearing the configuration memory units. Block RAMs are reset to their initial state, flip-flops are reset, and the entire memory is sequentially cleared.

A set of physical jumpers outside of the FPGA device (known as mode pins) are used to specify which configuration interface will perform the initial programming of the bitstream. These jumpers may be manually adjusted by the user before the device is powered on. These pins are sampled after the memory is cleared to indicate which interface will be used. The configuration interface issues the SYNC word (see Section 3.3.4), which alerts the configuration module to begin processing subsequent data words.

The configuration data is sequentially loaded into the FPGA, followed by a device-wide CRC check. The CRC check compares a calculated CRC value based on all of the bitstream data that was processed through the configuration module and a value that was computed by the bitstream generation tools. If the comparison is a match, then configuration enters the STARTUP sequence which allows all Block RAMs and Flip-Flops to begin changing state.

3.3.2 Configuration Interfaces

Since user FPGA designs differ greatly and have different application requirements, FPGAs are equipped with multiple configuration interfaces, enabling the user to select the interface that best fits the needs of their system. While the basic command sequences of scrubbing operations are universal across all interfaces, there are important differences that must be considered to accurately use each interface. The primary configuration interfaces are summarized below:

1. **JTAG**

   The Joint-Test Action Group (JTAG) interface is a common, easy-to-use, serial interface that supports the IEEE 1149.1 standard for Test Access Port and boundary-scan architectures. Configuration commands through the JTAG interface must be wrapped in special JTAG command headers and footers to sequence through the JTAG state

machine protocol, including access to the various JTAG registers. One of the main advantages of this interface is that the JTAG hardware ports are already built into almost all Xilinx FPGA boards, so using JTAG only requires the use of those four pins [1]. JTAG is also the highest priority configuration interface [6], and will always be able to access the configuration module regardless of the configuration mode pins (see Section 3.3.1). This interface can program multiple devices, but must do so in a serial chain, (not in parallel like SelectMAP). This interface is typically accessed from an external source, with clock rates up to 66 MHz [41].

2. **SelectMAP**

   SelectMAP is a parallel, high-bandwidth interface with a bi-directional data bus supporting data widths of 8, 16, or 32 bits [1]. This interface features the ability to configure multiple FPGAs in parallel and can be used with high-speed clock rates up to 100 MHz [42]. The main disadvantage of this interface is that a number of I/O pins (equal to the data bus length) must be reserved during configuration by SelectMAP, and thus are temporarily not available to the user design.

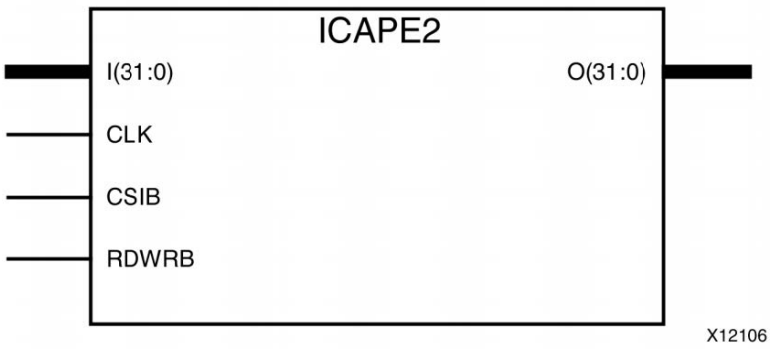

Figure 3.4: ICAPE2 Configuration Interface [2] pg. 187

3. **ICAP**

   Another option for performing configuration operations is to program the device from within the FPGA itself. The Internal Configuration Access Port (ICAP) can only be accessed from a user design via primitive instantiation (see Figure 3.4). The ICAP

allows a user design to reconfigure the device (either partially or fully), after the initial configuration has been completed. It is not able to perform the initial configuration, however. The ICAP is primarily used for partial reconfiguration. Like SelectMAP, one of its primary advantages is its configuration speed, an operating frequency of 100 MHz [3]. The ICAP is the interface of choice for several internal scrubbers discussed in Chapter 5.

4. **PCAP**

   The Processor Configuration Access Port (PCAP) is a unique interface that enables access from a hard processor to the configuration module. The PCAP is only found on the Zynq-7000 family. It is the bridge between the Zynq's dual-core ARM processing system and the FPGA configuration memory. One of its most important features is that it allows software programs running on the processors to access the configuration module at runtime via configuration commands written in software. The PCAP clock can run at frequencies as high as 500 MHz, though it usually runs at no higher than 100 MHz for most applications [43]. The PCAP will be discussed in greater depth in Chapter 7 and Appendix B.

Because the configuration module is the only gateway into the configuration memory, only one of the interfaces may actively process commands through it at any given time. A multiplexer function decides which interface controls the configuration module, so that that interface has exclusive control. The details of this multiplexer function are not publicly documented. A model based on experimental observations and the documentation is given here (see Figure 3.5), but it does not necessarily represent the actual implementation.

Some configuration interfaces exhibit an "access" priority: if one interface is using the configuration module, a higher priority interface can displace it and gain access even if the original interface is in the middle of a transaction. If a lower priority interface tries to use the configuration port while it is in use by a higher-priority interface, the request will be ignored. JTAG is the highest priority interface, and the Readback CRC is the lowest (see Chapter 4). It is unknown what the priorities of the other interfaces are. A higher priority

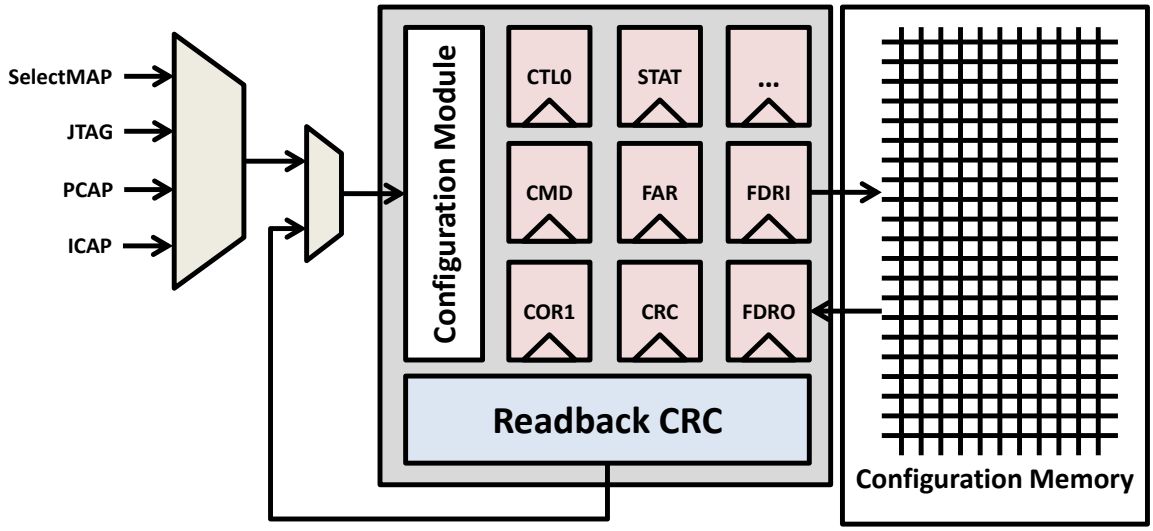

Figure 3.5: Configuration Interfaces and the Configuration Logic

interface must issue the DESYNC command to allow a lower-priority interface to gain access (see Section 3.3.4).

3.3.3 Configuration Registers

Inside the configuration module is a set of dedicated 32-bit hardware registers which handle all operations affecting the configuration. All configuration operations (see Section 3.3.5), are performed via reads and writes to these registers. These registers are not accessed directly from a user design, but instead through the configuration module. The read and write command sequences are sent through a configuration interface to the configuration module. While many configuration registers are listed in the documentation [1], only those pertinent to scrubbing will be described here (see Table 3.3). The registers listed here are those contained in 7-Series devices. The majority of these registers are the same as previous device generations, though the functionality of some has been changed in the 7-Series devices.

These registers fall into two different categories. Some registers are meant to only have certain bits modified which individually enable, disable, or alter specific configuration control features (e.g. CTL0, COR1). The remaining registers process their entire 32-bit word as a whole data value or command (e.g. FAR, CRC, IDCODE). Each of these registers will be summarized in the text below.

Table 3.3: Xilinx 7-Series Configuration Registers Used in Scrubbing [1]

Register Name	Category	Read/Write	Brief Description
FAR	Data	R/W	Frame Address Register
CMD	Data	R/W	Command Register
FDRI	Data	W	Frame Data Register Input (Write)
FDRO	Data	R	Frame Data Register Output (Read)
CTL0	Control	R/W	Control Register 0
COR1	Control	R/W	Configuration Options Register 1
MASK	Control	R/W	Mask Register for CTL0
STAT	Control	R	Status Register
IDCODE	Data	R/W	Device ID Register
CRC	Data	R/W	Bitstream CRC
CRC_HW	Data	R	Hardware-Generated Golden CRC
CRC_SW	Data	R/W	User Golden CRC
CRC_LIVE	Data	R	User-Enabled Generated CRC

FAR

The Frame Address Register (FAR) contains the FRAD of the frame that is currently being accessed by the configuration module. Any read or write operation to the configuration memory starts with the frame indicated by the address in the FAR. The FAR automatically increments to the address of the next frame as each frame is written or read, unless the end of a row has been reached (see Section 6.1.1). The number of frames incremented during a read or write is determined exactly by the number of words specified in the write or read operation, including any necessary dummy frames (see Section 3.4.2).

CMD

The command (CMD) register instructs the configuration module to perform specific functions. Important CMD commands include WCFG (write configuration data) and RCFG (read configuration data). Each precedes its respective write or read of the FDRI or FDRO. Another CMD command found in full bitstreams is the STARTUP sequence which is the last stage of the configuration process (see Section 3.3.1). The Reset CRC command resets the CRC check of the configuration process to start computing the CRC value from that point forward. The termination command used in configuration operations is the DESYNC

command, which releases the configuration port to allow a lower-priority interface to access it (see Section 3.3.4).

FDRI/FDRO

The Frame-Data-Register-In (FDRI) and Frame-Data-Register-Out (FDRO) are the ports into and out of the configuration memory respectively. A write to the FDRI is transferred to the configuration memory at the address specified in the FAR. Every configuration data word of the bitstream is written to this register in order to be sent to the actual configuration memory. Equivalently, every data word read from the FDRO comes directly from the configuration memory at the frame whose address is currently in the FAR. Preceding either of these scenarios, these registers must first be written with the number of words to be transmitted to or from the configuration memory.

CTL0/MASK

The Control 0 (CTL0) register specifies certain options used in configuration. This register is masked by the MASK register. In order to write to CTL0, the MASK register must first be written with the bits corresponding to the locations that the user wants to modify in CTL0. One of the most important bits of CTL0 is the Global LUT mask (GLUTMASK) (bit [8]). When enabled, the GLUTMASK bit causes any changeable memory cell readback values such as distributed RAM, SRLs, and DRP memories to be read back as all zeros or all ones [3] (see Section 3.4.1 for more information). This feature is important since the memory's real values change with normal functionality, and these changes would otherwise be detected as upsets during readback.

COR1

This register also sets important configuration options for the device, and is key to controlling the Readback CRC mechanism (see Chapter 4). The documented bits are given here, while others, which are reserved bits, are discussed in Appendix C.

1. **RBCRC_ACTION**

   (Readback CRC Action) The behavior of the RBCRC module is determined by these

two bits [16:15], which can be set to CONTINUE, HALT, CORRECT_AND_HALT, or CORRECT_AND_CONTINUE. The differences in these modes are discussed in Section 4.3.1.

2. **RBCRC_EN**
   (Readback CRC Enable) Bit [8] enables the Readback CRC mechanism to scan all of the frames via continuous readback.

STAT/IDCODE

The Status and IDCODE registers are primarily used for debugging and status checking. Each unique FPGA device has its own ID code. Writing the correct device ID to the IDCODE register is actually a necessary step when performing any writes to the configuration memory.

The status register is useful in determining certain errors such as decryption, Device ID, and CRC check errors. It also indicates the status of important start up components and signals including the mode pins.

CRC

The CRC registers are used in the device-wide CRC check and control the behavior of the CRCERROR signal (see Section 4.2.2). The CRC register described in the Configuration guide (not the ones mentioned in Appendix C), is solely used to check for bitstream integrity as described in Section 3.3.1. The footer of the bitstream contains the value of what the calculated CRC should be (as computed by the bitstream-generation tools). That value is written to this CRC register and compared with what the FPGA has just calculated after programming the bitstream. If a mismatch exists, then a CRC error has occurred.

The other CRC registers are used as a part of the Readback CRC mechanism (see Chapter 4). These registers are discussed in more detail in Appendix C.

3.3.4 Synchronization Commands

All data going into the configuration module is ignored until a special SYNC word is detected. The special SYNC word alerts the configuration module to begin processing all

subsequent data words (instead of ignoring them). On the other hand, issuing the DESYNC command stops the configuration module from accepting new data.

The Xilinx universal configuration SYNC word signals the configuration module to align at a 32-bit word boundary [1]. All commands presented to the configuration pins before the SYNC word are ignored. Thus, all configuration sequences begin with the SYNC word, or possibly a leading dummy word followed by the SYNC word. The SYNC word is `0xAA995566`.

The Desynchronization (DESYNC) command, processed by the CMD register, releases the configuration module allowing lower-priority interfaces to access it. The DESYNC command tells the configuration module to ignore any subsequent values coming in on the data pins (until another SYNC word is detected). The DESYNC command is also used to clear the CRCERROR signal (see Section 4.2.2).

3.3.5 Configuration Operations

All configuration operations are performed via reads and writes to the configuration module using a configuration interface. These configuration operations can be summarized into four categories:

1. **Write Configuration Data**

   This operation is nearly identical to the header of a full bitstream. However, like partial bitstreams, writes to the configuration may be just a few frames.

2. **Read Configuration Data**

   Reading configuration frames is also known as performing "readback." The main difference between reading from the configuration and writing is that reading requires a request to first be issued, followed by a reception of the requested data. The configuration interface must change directions, and the user design which issued the request must be ready to receive the incoming data. Section 3.4 gives more information about readback.

3. **Write Configuration Register**

   As mentioned in Section 3.3.3, the operations of configuration are handled through

a series of writes and reads to the internal configuration registers. All configuration registers operations in a full bitstream are writes; there are no reads.

4. **Read Configuration Register**

   Some configuration registers, such as RBCRC_LIVE and FAR, contain valuable information regarding the state of a configuration transaction. For debugging purposes, it is useful to have the ability to read these registers. Scrubbers will read configuration registers to confirm that configuration memory operations are processing normally. Reading the FAR, for instance, is necessary to generating a FRADs list (see Section 6.2.4).

Creating generic command sequence templates for the four types of configuration operations is not difficult. Using these templates, only the frame locations, number of frames involved, specific configuration register, etc. need to be specified with each successive operation. Example command sequence templates are given in Appendix D.

With a basic understanding of the configuration registers and synchronization commands, an example bitstream command sequence should be easy to comprehend. Table 3.4 gives a sample bitstream layout with explanations. Note that for clarity, this example omits some configuration registers and commands not discussed in this thesis. (For a complete bitstream layout example, see pages 100–103 of [1]). The purpose of including this bitstream example is to show the order of how each of the configuration registers is written in an actual bitstream.

3.4 Readback

Xilinx FPGAs allow users to read configuration memory data through a configuration interface. Read operations or "readbacks" are performed by sending a readback request command sequence to the configuration module and then receiving the data returned by the configuration memory via the same interface [1].

Much like a bitstream, a readback request command sequence is a series of writes to registers culminating with a read from the FDRO register (see Section 3.3.3 under **FDRO**).

Table 3.4: Bitstream Command Sequence Example

Command Word	Section	Description
0xFFFFFFFF	Header	Dummy Word
0xAA995566	Header	Sync Word
0x20000000	Header	NOOP
0x30008001	Header	Write CMD
0x00000007	Header	Reset CRC Check
0x20000000	Header	NOOP
0x3001C000	Header	Write COR1
0xXXXXXXXX	Header	Set COR1 bits
0x30018001	Header	Write IDCODE
0xXXXXXXXX	Header	Device ID
0x3000C001	Header	Write MASK
0xXXXXXXXX	Header	Set Mask bits
0x3000A001	Header	Write CTRL0
0xXXXXXXXX	Header	Set Ctrl0 Bits
0x20000000	Header	NOOP
0x30002001	Header	Write FAR
0xXXXXXXXX	Header	FRAD
0x30008001	Header	Write CMD
0x00000001	Header	WCFG
0x30004000	Header	Write FDRI
0x5XXXXXXX	Header	Number of Words
0xXXXXXXXX	Data	Word 0
0xXXXXXXXX	Data	Word 1
...	Data	Word i
0xXXXXXXXX	Data	Word n - 1
0x30000001	Footer	Write CRC Check
0xXXXXXXXX	Footer	CRC Value
0x30008001	Footer	Write CMD
0x00000005	Footer	STARTUP
0x30008001	Footer	Write CMD
0x0000000D	Footer	DESYNC

Examples of readback command sequences can be found in Appendix D and in [1] on pages 126–128.

Certain important features and nuances associated with readback are essential to know for scrubbing purposes. These features are the classification of individual bits as either masked or essential and the presence of dummy frames in readback data and bitstreams.

3.4.1 Bit Classifications

In addition to generating the full bitstream file, the Xilinx tools can also generate optional files which provide further information about individual configuration bits. These files classify bits as "masked" or "essential."

Masked Bits

The first classification type is the masked bit. Some configuration memory locations change during normal execution of a design. These bits contain the state of memory resources and include FFs, specialized LUTs (LUTRAM or SRL), distributed RAMs, and Dynamic Reconfiguration Port (DRP) memories [1]. Built-in mechanisms prevent these dynamic bits from being mistaken as errors in the configuration. When a normal readback is performed from these memory locations, the contents appear as all zeros or all ones (unless Readback Capture is used), while writes to these locations have no effect [3]. Bits which exhibit this behavior are known as masked bits. Using the GLUTMASK signal (see Section 3.3.3 under **CTL0**), some of these masked bits can be turned off (i.e. no longer masked) so that they can be manipulated like normal configuration bits. The majority of masked bits, however, are truly masked, preventing any access or modification regardless of the state of the GLUTMASK signal.

Xilinx specifies some of the masked bits of a given design in a file known as the MSK file. This file is optionally generated alongside the bitstream[2].

Essential Bits

The other bit classification is an essential bit. The bitstream-generation tools can indicate exactly which bits in the bitstream are associated with the circuitry of the user design [44]. These bits are classified as essential bits. Xilinx specifies these bits in a separate file optionally generated, like the MSK file, alongside the bitstream[3]. These bits are not necessarily the same as sensitive bits (discussed in Section 2.1.2), since essential bits are determined at compile time instead of runtime like sensitive bits [24]. However, knowing which

[2]MSK files are generated with the BitGen option: -m

[3]Essential bit (EBD) files are generated with the BitGen option: -g essentialbits:yes

bits are essential allows a scrubber to report that information when an upset is detected, and the essential bits file is much easier to obtain than independently determining all of the sensitive bits of a design.

3.4.2 Dummy Frames

A pad or "dummy" frame accompanies all reads and writes of configuration data. This dummy frame accounts for the frame buffer which is between the FDRO/FDRI and the configuration memory [1]. Not much is known about the frame buffer, but a dummy frame is required in every configuration memory read or write to flush it. For instance, if a user wants to readback exactly one frame, then they must request two frames. The first frame of data will be the dummy frame stored in the frame buffer, and the second will be the actual data. Thus, the user will always request at least one frame more than the number of desired frames to account for the frame buffer (see Figure 3.6).

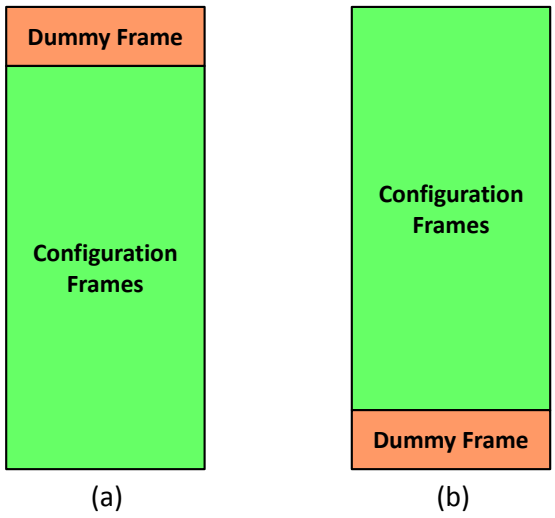

Figure 3.6: Location of the Dummy Frame in a (a) Readback and (b) Write To Configuration

In the case of writing, a dummy frame is appended onto the end of the rest of the configuration data being written. Thus, when the number of words to be written is sent to the FDRI register, the sequence must include an additional frame's worth of words to

process correctly. Read or write configuration operations without a dummy frame invariably fail. Note that dummy frames do produce complications for the auto-increment feature of the FAR (Section 6.1 describes these complications in a more lengthy discussion).

3.5 Partial Reconfiguration

Occasionally when only a small section of a design needs to be modified, a full reconfiguration is actually unnecessary. In fact, a full reconfiguration is quite disruptive because it resets all of the logic state of memory resources and restarts the device completely.

Partial reconfiguration is the ability to dynamically reprogram selected configuration frames while permitting the remaining resources in other frames to continue to operate without interruption [45]. Partial reconfiguration strictly targets a few frames in the configuration and generates a small "partial" bitstream to reprogram only those frames. The past several generations of Xilinx FPGAs support partial reconfiguration [42].

Partial bitstreams serve as the primary model for constructing scrubbing command sequences. Scrubbing operations are performed using many of the same commands as full and partial bitstreams, omitting or adding commands depending on the operation. Examples of these scrubbing command sequences are given in Appendix D. In addition, the ability of a scrubber to support partial reconfiguration (i.e., to not detect the changes of partial reconfiguration as errors) is a desirable feature.

The information presented in this chapter provides the configuration foundation upon which the scrubbers introduced later in this thesis are built. The process of scrubbing can be broken down into a series of reads and writes to configuration registers and then correctly interpreting the returning data. Scrubbing command sequences are patterned after full and partial bitstreams, including only the relevant commands for the scrubbing operation. This chapter, along with Appendix D, provides a thorough reference for basic scrubbing operations.

Chapter 4

SEU Configuration Memory Protection for 7-Series FPGAs

Newer FPGA families have developed increasingly advanced upset mitigation mechanisms with each subsequent FPGA family release. Preliminary upset mitigation mechanisms surfaced in devices as early as the Virtex-4 series [27]. The components evolved from generation to generation to become what the 7-Series family has today: a number of features and mechanisms for protecting the device against upsets in the configuration memory. These components include an individual ECC word for each configuration frame as well as a device-wide CRC mechanism. The individual frame ECCs enable SBU correction and two-bit upset detection, while the CRC detects nearly all MBUs. The most important feature of the ECC and CRC encodings is the hardware engine that actively uses them: the internal scan or "Readback CRC."

The Readback CRC continuously scans all of the configuration frame data, runs the ECC check for each frame, and runs the CRC comparison for the entire device. Its dedicated circuitry on-board the FPGA enables very fast MBU detection and SBU correction speeds. Properly using and interpreting the output of this internal scan requires careful attention to certain details which will be discussed in this chapter. This chapter presents the details necessary to use and understand the Readback CRC scan, including 7-Series configuration frame layout, the `FRAME_ECCE2` primitive, and the observed behavior of the scan itself.

4.1 7-Series Frame Layout

The purpose of this section is to explain in greater detail the organization of frames in the 7-Series FPGAs. While the topics of this section are diverse, they are all essential in understanding how individual frames are protected by the Readback CRC.

4.1.1 Frame Block Types

Each frame belongs to a specific block type based on bits [25:23] of its FRAD (see Figure 3.2). The block type indicates the functionality that the frame's data controls. There are five block types of frames in 7-Series FPGAs (see Section 6.2.4 for an example of how these different types were discovered). The largest block type representation is configuration or Type 0 frames. Table 4.1 gives the number and percentage of each frame type for an actual 7-Series FPGA. This data can easily be extracted from any FRADs list for a given part (see Section 6.2.4 and Appendix F).

Table 4.1: Zynq XZC07020 Frame Types Representation

Type	**Count**	**Percentage**
0	7692	74.1%
1	2304	22.2%
2	222	2.1%
3	18	0.2%
4	146	1.4%
Total	**10382**	**100%**

The scrubbing architectures developed by this thesis exclusively monitor Type 0, Type 2, and Type 3 frames, as these frames contain static data [3]. Type 1 (Block RAM) frames are dynamic, which means that their contents change with normal circuit function and thus cannot be scrubbed via a configuration scrubber. They must be protected using different upset mitigation methods such as TMR or ECC (see Section E.1).

4.1.2 ECC Word

Each frame has a built-in ECC word that is used by the Readback CRC to achieve SECDED functionality (as explained in Section 2.2.2). The Readback CRC initiates the computation of the ECC for each frame which computes a 13-bit syndrome value using all of the words of that frame. For the 7-Series, there are 100 data + 1 ECC = 101 words with

32 bits per word or 3,232 bits per frame. The 51st word in Types 0, 2, and 3 frames is the ECC word (see Figure 4.1).

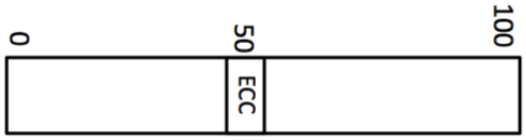

Figure 4.1: 7-Series Frame Layout

With a 13-bit syndrome there are $2^{13} = 8192$ possible syndrome values, or 8191 unique error representations. Since the syndrome can indicate the precise location of an SBU, 3,232 of these syndrome values must indicate unique bit locations. Since the code is a SECDED code, two-bit upsets can also be detected but not corrected. All odd-numbered MBUs will be detected by the ECC (though the syndrome value will not actually indicate any of the upset bits). The vast majority of even-numbered upsets larger than two bits will also be detected, but detection is not guaranteed in rare circumstances.

Bit [12] of the syndrome is the parity bit that is used to determine whether the detected upset is odd or even-numbered. If this bit is the only upset bit, then a "parity error" has occurred. The possible combinations of the rest of the syndrome bits and the parity bit are shown in Table 4.2. Recall from Section 2.2.2 that a satisfied ECC indicates no errors, or, in the very rare case that a large even-numbered upset has occurred in just the right locations so as not to be caught by the ECC.

Table 4.2: ECC Syndrome Scenarios

Type of Upset	**Parity Bit**	**SYNDROME[11:0]**
ECC Satisfied	0	Zero
Single-bit error	1	Nonzero
Parity error	1	Zero
Two-Bit error	0	Nonzero

It should be noted that these syndrome combinations are guaranteed for the upset types specified. They are also guaranteed for the majority of MBUs greater than two-bit upsets. If an odd-numbered MBU occurs, its syndrome will look just like that of a single bit upset. Similarly, the vast majority of even-numbered MBUs greater than two bits will have syndromes very similar to that of a two-bit upset (the exception is even-numbered MBUs which satisfy the ECC).

To illustrate the usefulness of the ECC decoding process, suppose that a 7-Series frame (101 words) has the contents shown in Table 4.3. For clarity and simplicity purposes, only the nonzero contents are shown (i.e. assume that all other words in the frame are 0). Note that word 50 (the 51st word) contains the embedded ECC golden value for the whole frame. This ECC word was calculated based on the nonzero data values in words 0, 2, 3, 4, and 5.

Table 4.3: 7-Series Frame Without Upset

WORD	BITS
0	0x4002040
2	0x100000
3	0x10
4	0x4000000
5	0x10
50	0x3BF

Now suppose that an SBU occurs at word 93, bit 10. The upset is reflected in Table 4.4. When the Readback CRC calculates the ECC for this frame which now has an upset bit, the resulting ECC word is `0x1CB5`.

The Readback CRC would then XOR the original and newly calculated ECC words which yields `0x3BF` $\oplus$ `0x1CB5` = `0x1F0A`. Thus, `0x1F0A` is the resulting syndrome value. Because the parity bit of the resulting syndrome equals one, an odd-numbered upset has been detected by the ECC. The word and bit information of the upset can be extracted from the value `0x1F0A` as instructed by [46] (see code snippet below).

Table 4.4: 7-Series Frame With Upset

WORD	BITS
0	0x4002040
2	0x100000
3	0x10
4	0x4000000
5	0x10
50	0x3BF
93	0x400

```
synbit <= syndrome[4 downto 0];
if (syndrome[12 downto 5] < X"A0") then
        synword <= syndrome[12 downto 5] - X"99";
elsif (syndrome[12 downto 5] < X"C0") then
        synword <= syndrome[12 downto 5] - X"9A";
elsif (syndrome[12 downto 5] <= X"FF") then
        synword <= syndrome[12 downto 5] - X"9B";
end if
```

This solution is not guaranteed to work for all cases, but generally the lower five bits of the syndrome give the location of the erroneous bit. The upper eight bits (including the parity bit) minus an offset give the location of the erroneous word. Thus, the results of the above example are as follows:

$$\text{error word} = SYNDROME[12:5] - \texttt{0x9B} = \texttt{0xF8} - \texttt{0x9B} = \texttt{0x5D} = 93$$
$$\text{error bit} = SYNDROME[4:0] = \texttt{0x0A} = 10.$$

The result of the syndrome comparison reveals the exact location of the upset, which can then be easily corrected by a repair mechanism like the Readback CRC. In the case of an odd-numbered MBU, the location given by the syndrome is very likely *not* one of the error bits. Instead, the syndrome gives the location of the bit that, when flipped, would satisfy

the ECC to produce an all-zero syndrome. Hence, if a three-bit upset occurs and the upset bit (indicated by the syndrome) is flipped, the result will be a four-bit upset, and the ECC will most likely not detect the error (because the ECC is now satisfied). In this case, more advanced mitigation techniques are necessary to handle these insidious upsets (see Section 4.3.2 for more examples of these types of upsets).

4.1.3 Frame Interleaving

In an attempt to split up MBUs between different frames to allow ECC coverage to have a greater error detection and correction, it appears that the configuration bits of adjacent configuration frames are physically interleaved [3], [47]. The purpose of this layout is to spread a two-bit upset in one frame to become two SBUs in different frames [16]. Since the Readback CRC mechanism easily handles SBUs, this frame interleaving is effective at reducing MBUs, but it by no means eliminates them. While the exact interleaving layout is not known, Figure 4.2 shows an example of how two adjacent frames might be interleaved.

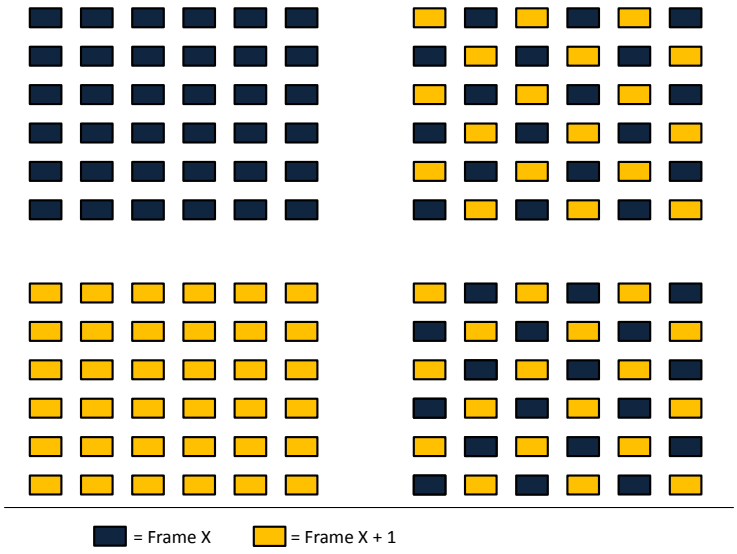

Figure 4.2: Uninterleaved Frames of Typical FPGA Families (Left); Interleaved Frames of the 7-Series (Right)

When a particle strike occurs affecting two adjacent bits in the uninterleaved frames, an MBU has occurred in that frame. On the other hand, when the same energy particle strike occurs in the interleaved frames and two adjacent bits are affected, the upset is spread across the two frames producing an SBU in each [48]. SBUs are much more easily correctable thanks to the frame's ECC, whereas MBUs are not able to be corrected by the ECC.

4.2 FRAME_ECCE2

The `FRAME_ECCE2` is a dedicated hardware primitive that is essential to many scrubbing architectures. There is one in each FPGA, and it enables the user or scrubber to monitor the built-in Readback CRC error detection and correction signals. The Frame ECC is inactive until a user performs readback through a configuration interface (see Section 3.3.2). Every readback, (regardless of the configuration interface), passes through the Frame ECC. The outputs of the Frame ECC then indicate whether the ECC calculation circuit found an error or not. Some 7-Series scrubbing architectures use the outputs (see Figure 4.3) of the `FRAME_ECCE2` primitive to detect upsets in the device. These signals convey valuable information about the locations, as well as the types of upsets that have occurred. The CRCERROR signal has a unique generation source that will be described in Section 4.2.2.

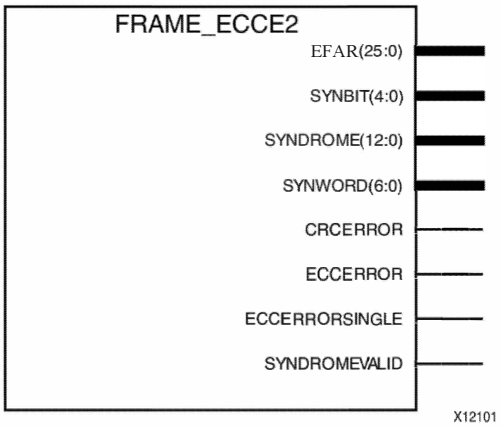

Figure 4.3: FRAME_ECCE2 Hardware Primitive [2] pg. 151

4.2.1 FRAME_ECCE2 Signals

This section gives a summary of each of the `FRAME_ECCE2` signals and what information they convey regarding the integrity of the configuration memory [33]. The generation of the CRCERROR signal is more complicated than the other `FRAME_ECCE2` signals and is described in its own section.

1. **EFAR (Error FAR)**

   This signal displays the value contained in the Frame Address Register (FAR), unless an upset has been detected. If an upset has been detected, then this signal holds the address of the frame where the error was detected, hence the name EFAR or "Error FAR." If multiple frames contain upsets, only the FRAD of the first-encountered (earliest address) even-numbered upset will be saved into the EFAR. Once all upsets have been corrected, the EFAR resumes displaying the current value in the FAR.

2. **SYNBIT**

   This signal holds the bit location of an upset if the upset is an SBU. This signal is derived from the SYNDROME signal. If there are an even number of upsets, then this signal will be zero (because location detection is not possible). It will also be zero if there are no upsets (because the SYNDROME is zero). If the error is an odd-numbered MBU, then this signal will be populated with a non-zero value that does not relate to the location of any of the upset bits, but instead is the value that satisfies the ECC (see Section 4.1.2).

3. **SYNDROME**

   This signal is the result of the ECC calculation performed on all of the bits in the frame, and subsequent comparison to the golden ECC word stored in that frame. A nonzero syndrome indicates that an error has been detected. See Table 4.2 for an explanation of how different types of errors are represented with this signal.

4. **SYNWORD**

   This signal holds the word location of an upset if the upset is an SBU. This signal is also derived from the SYNDROME signal. If there are an even number of upsets, then

this signal will be zero (because location detection is not possible). Like SYNBIT, if the error is an odd-numbered MBU, then this signal will be populated with a word location that satisfies the ECC but may not actually be one of the upset words. In addition, for 7-Series devices, this signal is 7 bits long, which means it can have $2^7 = 128$ possible values. Since there are only 101 words in a 7-Series frame, a SYNWORD value greater than 100 indicates that a special type of odd-numbered MBU has been detected (see Odd Out-Of-Bounds MBU in Section 4.3.2 for more information).

5. **CRCERROR**

   This signal indicates that the Readback CRC has detected a CRC error. Since the generation of this signal is more complex, it will be described in Section 4.2.2 and Appendix C.

6. **ECCERROR**

   This signal indicates that an ECC error has been detected. It is calculated by simply OR-ing all of the bits of SYNDROME. Hence, if any the SYNDROME bits are nonzero, then an ECC error has occurred.

7. **ECCERRORSINGLE**

   This signal indicates whether an upset is an SBU or even-numbered MBU. More specifically, it indicates whether the upset is odd-numbered (including SBU) or even-numbered. This signal is directly driven by SYNDROME[12], the parity bit.

8. **SYNDROMEVALID**

   This signal indicates that the value present on the SYNDROME pins is valid, which means that the ECC computation and comparison for that frame is complete. It pulses for exactly one clock cycle for each frame (though it takes multiple clock cycles to completely process an entire frame). Since ECCERROR, ECCERRORSINGLE, SYNWORD, and SYNBIT are all derived from the SYNDROME signal, they are all valid as well. The value on the EFAR signal is also valid insofar as the scan has not previously encountered an even-numbered MBU in an earlier frame (see the description

of the **EFAR** signal). CRCERROR is by no means guaranteed to be valid when this is asserted.

4.2.2 CRCERROR Signal

While the Readback CRC computes syndromes for each frame, it simultaneously contributes to a continuous a CRC calculation that terminates when the end of the device is reached. Each frame's data contributes to the CRC computation. The final CRC value (after scanning all of the frames) is compared to the golden CRC value held in a configuration register set during the initial passes of the Readback CRC (see Appendix C).

If there is a mismatch between between the golden CRC value and the calculated CRC value from the most recent scan of the entire device configuration, then the CRCERROR signal is asserted. One upset bit, if left uncorrected, could change the CRC to be an entirely new value, though it would first be caught and corrected by that frame's ECC before it would persist to a CRC error.

The CRC calculation is performed by XORing certain bit locations of a frame[1]. The 7-Series CRC code can detect up to a 31-bit upset in a single frame with 100 percent accuracy (see Section 2.2.3), and the accuracy is near 100 percent for larger upsets [49]. SEUs causing MBUs of this magnitude are not realistic; thus, the probability is very low that upsets can break both the ECC and CRC codes and go undetected.

The CRCERROR output is essential to scrubbing because it can distinguish odd-numbered MBUs from SBUs. Both of these types of upsets generate syndromes that appear identical (i.e. ECCERRORSINGLE is high for both). Thus, odd-numbered MBUs alias as SBUs. Even-numbered MBUs generate a different syndrome that is always distinguishable from an odd-numbered upset's syndrome (i.e. ECCERRORSINGLE is low). Table 4.5 gives the values of these two `FRAME_ECCE2` signals for different upset types.

After the Readback CRC finishes the CRC computation with the last frame, the CRC comparison is performed and the value is loaded into a comparison (LIVE) register [50]. This loading triggers a comparison between the calculated LIVE register and the

[1]Refer to the file FRAME_ECCE2.vhd v.11.1 (included in Xilinx Vivado software tools distributions) for the CRC circuit description for Xilinx 7-Series devices

Table 4.5: FRAME_ECCE2 Signals for Different Upset Types

Type of Upset	ECCERRORSINGLE	CRCERROR
SBU	1	0
Even-Numbered MBU	0	1
Odd-Numbered MBU	1	1

golden CRC register. If a difference exists, the CRCERROR signal is asserted. Since odd-numbered MBUs alias to SBUs, the CRCERROR signal is usually the only way to tell the two types apart.

One of the challenges of designing a reliable scrubber that uses the Readback CRC mechanism for error detection is capturing the CRCERROR output. This signal is unique in that its arrival comes many cycles[2] after the pulsing SYNDROMEVALID signal corresponding to the frame with the MBU. The `FRAME_ECCE2` signals should only be read simultaneously as a single batch of concatenated signals. The CRCERROR signal, however, comes only after the Readback CRC scans through the entire device. Thus, this signal's arrival is unpredictable and must be captured using an asynchronous latch independent of the other `FRAME_ECCE2` signals.

Not only is latching the CRCERROR signal separately somewhat cumbersome, linking it to the correct upset is also a formidable task. If multiple upsets have been detected and are waiting to be processed, it is unknown which one caused the CRC error. Scrubbing one of the upset frames could yield a useless scrub in that no MBUs were actually corrected (the Readback CRC has already corrected the SBU that was in that frame). The actual MBU remains uncorrected in a different frame.

A user can issue the DESYNC command (see Section 3.3.4), after the CRCERROR signal goes high, to clear the CRCERROR signal [1]. However, if the MBU is not fixed in the meantime, the next full scan of the Readback CRC will cause CRCERROR to go high again. If, after correcting an MBU and issuing a DESYNC CRCERROR goes high again, then another MBU still exists in a different frame.

[2]The Readback CRC circuit runs on an internal oscillator clock. However, instantiating the ICAPE2 primitive will change the Readback CRC's clock source [1].

4.3 Readback CRC

While CRCs have been used in Xilinx FPGAs for several generations, the Readback CRC or autonomous internal scan hardware mechanism was not introduced until the Virtex-5 device family [28]. The CRC polynomial implemented by the Readback CRC is the CRC32C checksum algorithm which has been used since the Virtex-4 family [27]:

$$x^{32}+x^{28}+x^{27}+x^{26}+x^{25}+x^{23}+x^{22}+x^{20}+x^{19}+x^{18}+x^{14}+x^{13}+x^{11}+x^{10}+x^{9}+x^{8}+x^{6}+1. \quad (4.1)$$

The early Readback CRC mechanism only calculated the device-wide CRC to detect errors. The frames of Virtex-5 FPGAs had ECCs as well, but no internal controller actually corrected any upsets; it was left to the user to create designs that performed the readback and corrected the upsets.

Virtex-6 devices introduced a greatly improved Readback CRC module, which the 7-Series families would later inherit [47]. Instead of requiring the user to create a design that performed continuous readback of the device so that the Frame ECC could check for errors, the new Readback CRC hardware automatically performed the readback scanning.

The only frames that are checked by the Readback CRC hardware are Types 0, 2, and 3 frames [3]. Type 1 (BRAM) frames and Type 4 frames are skipped entirely by the Readback CRC [1].

4.3.1 Readback CRC Action

The Readback CRC requires exclusive access to the configuration module. However, it is also the lowest priority interface (see Section 3.3.2). For the Readback CRC to run, all other configuration interfaces must have issued a DESYNC command to release the configuration module (see Section 3.3.4). Then the Readback CRC will automatically run depending on the POST_CRC_ACTION mode specified in the User Constraints File (UCF).

These constraints cause the bitstream generation tools to insert a write command to the COR1 in the bitstream header with the corresponding bits set (see Section 3.3.3 under

COR1). The POST_CRC_ACTION[3] constraint controls the mode of the Readback CRC mechanism [51]. It has four operating modes that may be useful depending on the needs of the user design:

1. **CONTINUE**

   If an ECC error is detected in the current frame, the scan continues on to the next frame while asserting the corresponding error signals (ECCERRORSINGLE, SYNBIT, SYNWORD, etc.) along with CRCERROR (regardless of the error type). No correction of SBUs is performed, however.

2. **HALT**

   If an ECC error is detected in the current frame, readback ceases, and the corresponding error signals are asserted. Note that CRCERROR will always be asserted in this mode, regardless of the error type detected.

3. **CORRECT_AND_HALT**

   If an ECC error is detected, the corresponding error signals are asserted, SBUs are corrected, and readback ceases. Note that CRCERROR will always be asserted in this mode, regardless of the error type detected.

4. **CORRECT_AND_CONTINUE**

   If an ECC error is detected, the corresponding error signals are asserted, SBUs are corrected without asserting CRCERROR, and readback restarts from the beginning of the device to continue scanning frames.

Each of the modes has different behavioral side effects with the best mode depending on the circumstances of the system. If the reliability requirements of the system are limited to error detection, then HALT or CONTINUE may be preferable. The CONTINUE mode is also preferable when debugging to ensure that fault injection (manually writing upsets into the configuration memory) works correctly (i.e. the injected upsets are not immediately corrected). HALT and CORRECT_AND_HALT could also be used for simpler correction

[3]UCF entry example: CONFIG POST_CRC_ACTION = CORRECT_AND_CONTINUE;

schemes that prefer to handle one upset at a time (avoiding many of the upset scenarios discussed in Section 4.3.3).

The CORRECT_AND_CONTINUE mode is the option used by the scrubbing architectures in this thesis that use the Readback CRC since the Readback CRC handles SBUs and continues scanning afterwards. Furthermore, all of the other three modes assert CRCERROR regardless of the error type (i.e. it asserts for SBUs as well as MBUs) [1]. Since the ECCERROR can identify SBUs efficiently, it is desirable to limit CRCERROR assertion to MBUs. This policy enables scrubbers to differentiate SBUs and odd-numbered MBUs accurately.

4.3.2 Upset Types

SEUs occurring in configuration memory typically affect only a single bit in a single frame. It is possible, however, that an SEU causes multiple bits in the same frame to be upset. Almost as likely is the case where an SEU causes multiple upsets in different frames (typically in adjacent frames). It is also conceivable that different SEUs occur simultaneously (or close enough such that the Readback CRC does not detect them separately) so as to produce upsets in different frames. All of these upset scenarios can be potentially difficult for a scrubber to both detect and correct accurately.

When an SEU causes bit upsets in a single frame, it is important to understand the severity of the upsets in order to know how to most efficiently correct them. For this thesis, the upset type is defined to be the number of bits upset in a single frame. Because each frame has its own ECC, the Readback CRC considers ECC upset detection and correction on an individual frame basis. Recall from Section 4.1.3 that SEUs can cause upsets in multiple frames. These separate upsets are treated as distinct events. Some upset types can combine to cause interesting behavior by the Readback CRC that will be discussed in Section 4.3.4.

The basic upset types are the SBU (one upset bit), the even-numbered MBU ($2n$ upset bits where $n > 0$), and the odd-numbered MBU ($2n + 1$ upset bits where $n > 0$). Odd-numbered MBUs can further be subdivided into in-bounds and out-of-bounds MBUs. An odd-numbered in-bounds MBU means that the syndrome value, though not actually indicating any of the real upset bits, is still holding a valid word number (0–100 for 7-Series).

An odd out-of-bounds MBU has a SYNWORD that holds an invalid word number (101–127). This type of upset is detected instantly by observing the SYNWORD value. At that point, there is no question that the upset is an odd-numbered out-of-bounds MBU (i.e. no need to wait for a CRCERROR to confirm). This upset type can happen only if *1*) the upset is an odd-numbered upset (since SYNWORD and SYNBIT do not populate with even-numbered upsets); and *2*) the upset is not an SBU. Otherwise, the SYNDROME would hold the value of an in-bounds upset bit. It is unknown how many of the possible combinations of odd-numbered MBUs would generate a SYNDROME with an out-of-bounds location; however, it would be expected that with a random upset distribution, about $27/128 = \sim 21\%$ of odd-numbered MBUs would be out-of-bounds.

Another rare type of upset that is harder to detect is an odd-numbered MBU with an attempted correction at a masked bit. Unlike the odd-numbered, out-of-bounds MBU, the SYNWORD in this upset type holds a valid, in-bounds word. The difference is that the actual bit in that word is masked (and therefore not writable). Masked bits are always read back as all zeros or all ones (see Section 3.4.1), so for a SYNDROME to indicate that one of these masked bits is now upset, means that the error must be an odd-numbered MBU.

4.3.3 Multiple Frames with Upsets

When more than one of these upset types occur in different frames by the same SEU or by different SEUs at nearly the same time, the Readback CRC exhibits peculiar behavior. Section 4.3.4 will discuss how the Readback CRC reacts in these situations. A list of the more common examples of these situations is given below. These scenarios were observed by performing constrained-random fault injection and in radiation testing. Note that an arrow ($\rightarrow$) means "followed by", (i.e. the first upset occurs at a lower FRAD, (the Readback CRC will detect it first), and is followed by a second upset occurring at a higher FRAD). Table 4.6 gives an illustration of these scenarios.

1. **Even-Numbered MBU $\rightarrow$ Odd-Numbered MBU/SBU**

   An even-numbered MBU occurs in a lower-address frame, and an odd-numbered MBU or an SBU occurs in a higher-address frame. The odd-numbered MBU may be in or out of bounds. See Table 4.6 for a visual representation.

Table 4.6: Two Common Combinations of Upset Types

FRAD	Even → Odd	Even → Even
0x0000000		
...		
0x900	Even Upset	Even Upset
0x901	Odd Upset	Even Upset
...		
0x3FFFFFF		

2. **Even-Numbered MBU → Even-Numbered MBU**

   Two or more even-numbered MBUs occur: one in a lower-address frame, and the other in a higher-address frame. See Table 4.6 for a visual representation.

The upset situations described in this section are not an exhaustive list of all possible combinations of upsets that could be seen in a radiation environment (there are likely more that are not currently known). They are presented here as a representative sample in preparation for the next section that will elaborate on the unexpected behavior of the Readback CRC when encountering these types of upsets. Section 4.3.4 will further describe what makes each of these upset scenarios unique with regard to the behavior of the Readback CRC hardware.

4.3.4 Readback CRC Behavior

As described in [1], the Readback CRC hardware behaves predictably with each upset type that occurs (see Section 4.3.2). The behavior, however, does not always convey what is really happening to the FPGA configuration memory. Based on documentation and on experimental observation, six important behaviors of the Readback CRC are summarized below. Note that these behaviors assume that the Readback CRC Action is CORRECT_AND_CONTINUE. This is the mode used by the scrubbing architectures developed in this thesis that use the Readback CRC. This summary is followed by a discussion of which upset types evoke which behaviors.

1. If the upset is odd-numbered ($2n + 1$ upset bits, where $n \geq 0$), the Readback CRC always attempts to correct the bit based on the location indicated by SYNWORD

and SYNBIT. This location may or may not be in-bounds. Following the attempted correction, the Readback CRC starts over at the first FRAD (usually `0x000000`). It does not continue on to the next FRAD.

2. If the upset is even ($2n$ upset bits, where $n > 0$), then the Readback CRC attempts no correction, and continues on to the next frame. In addition, once an even-numbered upset has been detected, SYNWORD and SYNBIT are set to zero and cease to update for subsequent frames that have errors until the even-numbered upset is corrected or undetectable by the ECC.

3. Upon reaching the last frame of the scan, if an MBU in any frame exists, then the newly calculated CRC value will differ from the stored golden CRC value. The result of this comparison causes the CRCERROR signal to be asserted.

4. If upsets in different frames exist, the EFAR only latches the FRAD of the first-encountered even-numbered MBU (i.e., the MBU at the lowest address). Odd-numbered upsets cause the Readback CRC to restart back to the beginning (see behavior #1), so the Readback CRC will be unaware of any upsets in later frames until all leading odd upsets are handled first. After the Readback CRC detects the even-numbered upset and continues on to the next frame (see behavior #2), the SYNDROME will continue to change, reflecting the ECC status of each frame. However, the EFAR maintains holding the FRAD of the first-encountered even-numbered upset until the scan is restarted. The EFAR will resume updating until it reaches the frame with the even-numbered upset again. Only by correcting the frame with the even-numbered upset will the EFAR begin again to update past the frame that had the even-numbered upset.

5. Odd-numbered MBUs cause an incorrect repair during the Readback CRC's attempt at correction. The incorrect repair occurs in the frame that the EFAR has latched. If SYNWORD is an in-bounds word, this attempted correction causes the odd-numbered MBU to become an even-numbered MBU. Oftentimes, because the incorrect repair occurs at the bit location that satisfies the ECC, future scans of the Readback CRC will

not detect this now-even-numbered MBU by the frame's ECC. The upset is completely undetectable by the ECC but will be detected by the CRC.

6. When the odd-numbered out-of-bounds MBU (described in Section 4.3.2) occurs, the Readback CRC's attempted correction fails because SYNWORD is an out-of-bounds location (the configuration frame actually remains unchanged). However, because a correction was attempted, the Readback CRC starts over from the first frame address of the entire device. When the Readback CRC encounters the frame again, the same ECC is calculated to again indicate an out-of-bounds location, and the Readback CRC will start over again from the beginning after attempting its correction. The obvious problem is that the Readback CRC enters an infinite loop, and, since the last frame is never reached, CRCERROR will never be asserted. Identical behavior is observed if SYNWORD and SYNBIT indicate a masked location within the frame.

Table 4.7 displays the observed Readback CRC behaviors for each upset type. The columns of this table represent the behaviors with the corresponding numbers in the list above. All behaviors that could happen (not necessarily together) in the given upset scenario are recorded. Behaviors not marked for certain upset types can never occur.

Table 4.7: Readback CRC Behaviors Based on Upset Scenarios

Upset Scenarios	**1**	**2**	**3**	**4**	**5**	**6**
SBU	X					
Even MBU		X	X			
Odd MBU (In bounds)	X		X		X	
Odd MBU (Out of bounds)	X					X
Even → Odd MBU/SBU	X	X	X	X	X	X
Odd MBU Correction at Masked Bit	X					X
Even MBU → Even MBU		X	X	X		

Oftentimes, multiple behaviors combine to escalate the difficultly of accurately diagnosing an error. For instance, suppose upsets occur in two different frames, and the first upset is an even-numbered MBU that causes the EFAR to be latched. If the second is also

an even upset, then behaviors 2 and 4 combine to create a situation that is difficult for a scrubber to detect and correct both upsets. The EFAR will only latch the address of the first upset (preventing the frame location of the second upset from being discovered), while the Readback CRC will continue to scan past each upset indefinitely, without correcting either one (because both upsets are even-numbered).

Scrubbing architectures which rely entirely on the Readback CRC will at best be unable to correct MBUs, and at worst be unable to detect them at all. However, because the Readback CRC mechanism is dedicated hardware positioned in close proximity to the configuration logic, it will still correct SBUs faster than other external scrubbing mechanisms and efforts to supplement this behavior will result in much faster scrub times.

Understanding the limitations of the Readback CRC hardware exposed by these upset scenarios is crucial to creating a more robust scrubbing architecture. The architecture can be designed to harness the performance advantages offered by the Readback CRC hardware for SBU correction, while compensating for its limitations with other mechanisms to attain the desired level of robustness.

4.3.5 Simultaneous Upset Example

To better illustrate the need for additional scrubbing support to help the Readback CRC in the case of multiple upsets in different frames, an example of an Even-Numbered $\rightarrow$ Odd MBU/SBU will be demonstrated in this section. Suppose that an SEU causes a simultaneous two-bit upset at FRAD `0x8A0` and an SBU at FRAD `0x8A1` (see Figure 4.4). The SEU caused upsets in different frames because of the physical interleaving of the frames (see Section 4.1.3). The Readback CRC hardware will report values similar to those shown in Table 4.8 on the `FRAME_ECCE2` signals.

The Readback CRC scans the frames sequentially and will encounter the two-bit upset first (syndrome: `0x73C`). Because it is even, the scan will continue, recognizing that it cannot correct the upset. CRCERROR, however, is not yet asserted because the end of the device has not yet been reached. Furthermore, the EFAR will latch the FRAD of the two-bit upset (`0x8A0`) and will not change until that upset is fixed.

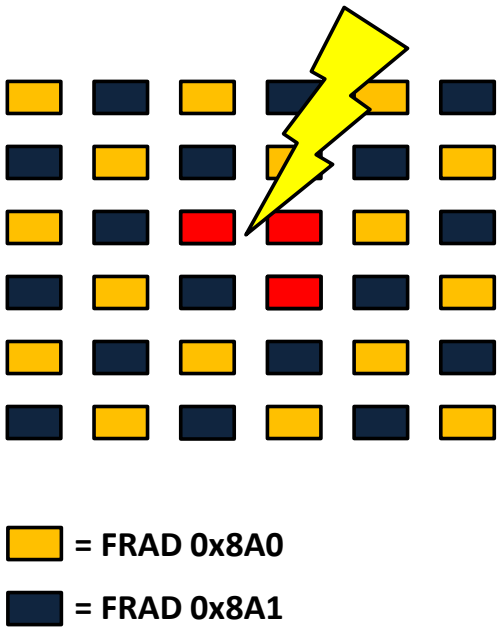

Figure 4.4: Two-bit Upset and SBU in Adjacent Frames

Table 4.8: `FRAME_ECCE2` Values for Simultaneous Even, Single Upset

#	EFAR	SINGLE	SYNDROME	SYNWORD	SYNBIT	ECC	CRC
1	0x8A0	0	0x73C	0	0	1	0
2	0x8A0	1	0x1842	0	0	1	0
3	0x8A0	0	0x73C	0	0	1	0
4	0x8A0	1	0x141C	0x06	0x1C	1	0
5	0x8A0	1	0x141C	0x06	0x1C	1	0
6	0x8A1	1	0x1842	0x27	0x2	1	0
7	0x8A1	1	0x1842	0x27	0x2	1	1

The scan continues and encounters the SBU in the next frame. The computed syndrome (`0x1842`) tells the Readback CRC hardware that this is an odd-numbered, and hence, correctable upset. The EFAR, however, is still locked onto the previous FRAD which still contains the even-numbered upset. Since the upset was even-numbered, SYNWORD and SYNBIT will be zero so long as the even-numbered upset remains uncorrected and does not satisfy the ECC. Furthermore, because the EFAR is still holding the previous FRAD, a bit is incorrectly repaired in the frame already containing the two-bit upset resulting in a three-bit upset at address `0x8A0`. The SBU still remains unfixed in frame `0x8A1`.

The scan restarts from the beginning of the device and encounters the now three-bit upset and computes a new syndrome: `0x141C`. Since the upset is odd-numbered, it also appears to be correctable, so another incorrect repair occurs in frame `0x8A0`, causing the three-bit upset to become a four-bit upset. The scan restarts a second time, but when it encounters the four-bit upset, the ECC is now satisfied, and a syndrome of all zeros results (note that not all incorrect repairs will satisfy the ECC). The scan then continues and finally reaches the real SBU in FRAD `0x8A1` and generates the syndrome of `0x1842`. Because no even-numbered MBU has been detected during this pass of the scan, no other EFAR is latched. The SYNWORD and SYNBIT will now update with the values from the syndrome and the Readback CRC will successfully correct the SBU. The scan restarts a third time and eventually will reach the end of the device at which point CRCERROR will go high, detecting the four-bit upset in FRAD `0x8A0`.

While it may seem tedious for a scrubber to watch for such unexpected behavior, doing so provides the scrubber with enough information to intelligently scrub FRAD `0x8A0` rather than reconfiguring the whole device, an inefficient fail-safe that earlier hybrid scrubbers depended upon [16]. It also enables recognition of the fact that the SBU in FRAD `0x8A1` was successfully corrected, and that no further scrubbing action is required. If, after correcting the MBU in FRAD `0x8A0` with an external scrubbing source, CRCERROR had gone high again with no further information displaying on the `FRAME_ECCE2` signals, then a full reconfiguration would be required. For this thesis, this type of error is a CRC-Only error.

Up to this point, it may seem that the limitations of the Readback CRC give cause for concern in its use of upset mitigation. However, since the vast majority of radiation upsets are SBUs (see Chapter 7), the Readback CRC is quite adequate and desirable since it is already built-in to the FPGA hardware. Furthermore, the competitive speeds of upset detection (both SBUs and MBUs) are among the fastest available for the 7-Series [3]. Harnessing these advantages is the goal of the scrubbing architectures of this thesis. To effectively utilize the Readback CRC, its limitations must be well-understood so that work-around solutions can be created.

This chapter described important 7-Series hardware mechanisms that are used in upset mitigation primarily consisting of the Readback CRC. This chapter also provided an illustration of how the Readback CRC interprets unique upset scenarios. Understanding the behavior of the Readback CRC and `FRAME_ECCE2` hardware components is essential to comprehending the design of the hybrid scrubbing architecture presented in Chapter 6.

Chapter 5

FPGA Scrubbing Techniques

Configuration scrubbing is the process of detecting and correcting upsets in the configuration memory of an FPGA. Over the past several generations of Xilinx FPGAs, many scrubbing techniques and architectures have been developed which have been validated by radiation tests and/or fault injection experiments. The purpose of this chapter is to review many of these scrubbing solutions for the Virtex-4, Virtex-5, and Virtex-6 families. Understanding the strengths of these scrubbers can aid in developing new and improved scrubbing architectures for more recent FPGA families.

This chapter begins by giving an overview of basic scrubbing concepts. The next section presents explanations of an important characteristic of scrubbers whether they are internal or external to the FPGA they are scrubbing. That section is followed by a summary of two of the most prevalent scrubbing strategies: blind and readback. This chapter concludes with several examples of scrubbing techniques and architectures developed previously including the predecessors of the hybrid scrubbing architecture, the primary contribution of this work.

5.1 Scrubbing Basics

Scrubbers require three basic components to performing scrubbing operations (see Figure 5.1 for a visual representation). First, scrubbers need access to the configuration module through a configuration interface. Second, scrubbers require some type of processing logic that forms the read and write command sequences and initiates data transfers to and from the configuration memory. This logic must also accurately interpret the data returned by the configuration module to make intelligent scrubbing decisions. Oftentimes, this logic is implemented by using a processor (hard or soft), or a Finite State Machine

(FSM) implemented in the FPGA logic. Finally, if more robust correction strategies are employed, then some sort of memory must hold the golden bitstream data used to overwrite upsets. This memory is typically a BRAM, external DDR or flash memory.

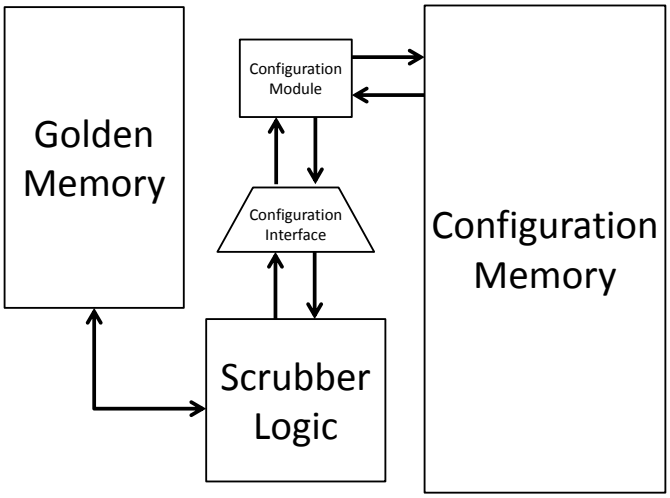

Figure 5.1: Basic Scrubbing Components

In addition to scrubbing components, scrubbers have a few important performance metrics that will be defined here. The majority of scrubbers systematically scan each frame of the configuration memory. Typically, the scanning is deterministic (see Section 2.2.6), meaning that each frame is scanned in consecutive order continuously. However, some scrubbers probabilistically scan frames in a more "random sampling" fashion. The performance metrics described below represent characteristics of deterministic scrubbers.

It is difficult to characterize scrubbing performance from the time of upset occurrence since it is unknown when the upsets will occur. For this reason, the performance metrics are defined here in terms of upper bounds. The total scrubbing time, T_s, is defined to be an upper bound from the time of upset occurrence until the upset is corrected.

Since upset correction is often preceded by upset detection, it is useful to define two separate metrics for these latencies. Upset detection latency, T_d, is the time to detect an upset in a single frame. Since detection is most commonly performed with some form of

readback, T_d is usually the time it takes to read back and check one frame. Upset correction latency, T_c, is the time to correct an upset (typically the time to write one frame).

The total scrubbing time can be written in terms of the detection and correction latencies:

$$T_s = N * T_d + u * T_c, \tag{5.1}$$

where N is the number of frames of the target FPGA, and u is the number of frames that had upsets detected in them. It is often useful to characterize detection performance with a single-upset upper bound detection latency, T_D, where only one upset exists on the device. In a worst-case scenario, the upset occurs in the most recently checked frame, which typically means that the entire device will have to be scanned before that frame is reached again. Therefore,

$$T_D = N * T_d. \tag{5.2}$$

5.2 External and Internal Scrubbers

All configuration scrubbing architectures require some means of access to the configuration memory (see Section 3.3). Configuration interfaces each have a configuration port that can access the configuration module, (see Section 3.3.2). Some interfaces are exclusively internal, exclusively external, or can operate as either type [17]. There are small advantages and disadvantages to both, but the preference of one over the other depends on the needs of the user design. These needs could include the availability of additional hardware to host a scrubber, the desire to separate the scrubber from the configuration memory being scrubbed, or the desire to increase the speed of the scrubber.

Internal scrubbers, or "self-scrubbers", are considered internal because no external hardware is required for their implementation. Typically, internal scrubbers use the Internal Configuration Access Port (ICAP) with a hardware state machine that employs a specific scrubbing strategy. Scrubbers which use the PCAP interface (see Chapter 7) are also considered internal. Internal scrubbers are *usually* faster than external scrubbers due to their close physical proximity to the configuration module. They also do not require the use of relatively long cables to connect the FPGA being scrubbed to the device running the scrub-

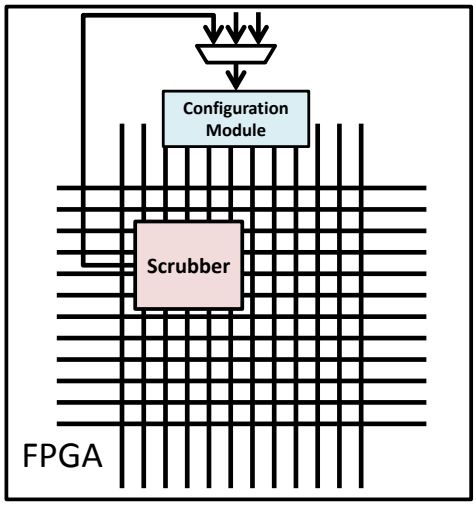

Figure 5.2: Example Layout of Internal Scrubber

ber. Internal scrubbers benefit by not needing another FPGA or microprocessor to host the scrubbing program; the entire scrubbing architecture is encapsulated within the FPGA itself (see Figure 5.2). Internal scrubbers do, however, require a portion of the FPGA logic to be reserved for the scrubbing hardware, which means that there are fewer available resources for the user design. Internal scrubbers are also just as susceptible to SEUs and common-mode failures as the configuration memory logic that is being scrubbed.

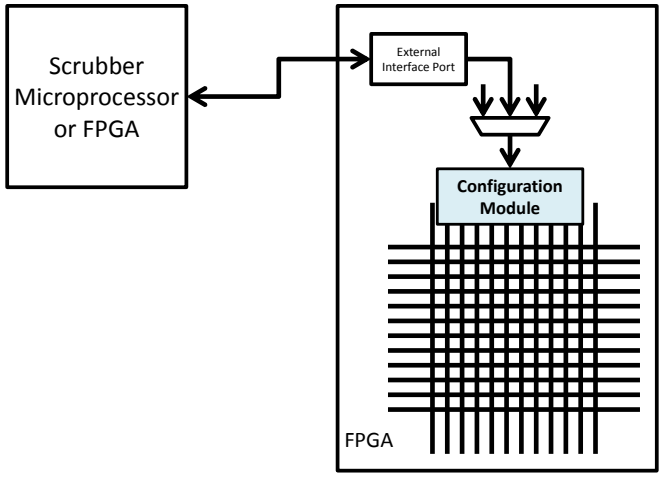

Figure 5.3: Example Layout of External Scrubber

External scrubbers typically use the JTAG interface, though scrubbers configured to use the SelectMAP interface can be either internal or external [42]. A separate FPGA or microprocessor implements the actual scrubbing logic for the external case (see Figure 5.3). Having an additional device to perform the scrubbing increases the power consumption of the system. External scrubbers usually connect into a port on the target FPGA board via circuit board traces or, in less common instances, variable-length cables. This fact is one of the reasons that external scrubbers must operate at lower clock frequencies than internal scrubbers. The primary advantage of external scrubbers is that they are more reliable since they are placed outside and away from the FPGA that they are scrubbing, which allows them to avoid common-mode failures in an environment such as a radiation beam test.

5.3 Common Scrubbing Strategies

Two of the most prevalent configuration scrubbing strategies are blind scrubbers and readback scrubbers. These strategies have trade-offs in terms of implementation size, performance, robustness, and efficiency. This section will present general overviews of these two strategies as well as highlight important differences between the two. Section 5.4 will cite examples of actual scrubbing architectures, some of which employ these strategies.

5.3.1 Blind Scrubbing

The blind scrubbing strategy is simply to continuously write to the configuration memory from a protected golden file. Blind scrubbing does not perform any detection ($T_d = 0$), and it assumes that every frame can have an upset ($u = N$). The total scrubbing time for the blind strategy is

$$T_{s_{blind}} = N * T_c, \tag{5.3}$$

where T_c is the latency of writing one frame to the configuration.

Blind scrubbing is simple to implement (see Section 6.3). Blind scrubbing also has fast correction speeds, at the cost of not having any detection capabilities. The blind scrubber can write one or more frames at a time (with optional delays in between), or write all of the configuration frames in one transaction. There may be instances where it is desirable

to insert delays (accounted for with the T_{delay} parameter) between each frame write to ease processing bandwidth resources. If delays are used, then

$$T_{s_{blind}} = N * (T_c + T_{delay}). \tag{5.4}$$

This type of frame writing is different from a full reconfiguration in that no STARTUP, shutdown, or CRC directives are included; it merely writes each frame continuously with the contents of the golden file.

One of the main disadvantages of blind scrubbing is its inherent inability to detect upsets. Since no configuration readback is occurring, nor any checking of the configuration status, no upset detection is possible (apart from recognizing that the design has malfunctioned, i.e., incorrect outputs are observed or an invalid state transition occurs). Moreover, a significant portion of processing bandwidth is wasted in continuously correcting already valid data [42], making it comparatively inefficient.

5.3.2 Readback Scrubbing

The readback scrubbing strategy repairs upsets by reading back the contents of the frame, checking to see if upsets exist, and overwriting the contents of the frame with the correct data. Like blind scrubbing, the scrubber moves through the configuration frames sequentially and then restarts back to the beginning of the device when the end has been reached. The worst case scrubbing time for readback scrubbing is

$$T_{s_{readback}} = N * T_d + u * T_c, \tag{5.5}$$

where T_d is the amount of time to read back and compare one frame and T_c is the amount of time to write one frame. When only one upset occurs at a time (i.e. between corrections), then $T_D = N * T_d$.

Upon reading a single frame, the detection of errors within the frame is accomplished by one of two methods [42]:

1. **Golden Data**

   A large memory can be used to store the golden data (see Figure 7.1 for an example), which is a copy of the initial configuration data loaded onto the device (see Section 6.2.1). After reading back a frame, the data can be compared word by word to its corresponding location in the golden memory, and, if a discrepancy exists, an upset has been detected. Typically, this memory needs some sort of upset mitigation protection such as ECCs or radiation-hardened fabrication to ensure proper functionality.

2. **ECC**

   Whenever a readback of a frame is performed (regardless of the interface), an ECC calculation occurs over the readback data. The ECC result can be compared to a stored ECC value to detect if an upset has occurred (see Section 2.2.2). The ECCs are stored either in a separate memory (like the golden data, though a lot smaller), or in the frame itself (like the Xilinx FPGAs; see Section 4.1.2). The Frame ECC primitive is often used to read the result of the ECC calculation performed during the readback.

Once an upset has been detected, the frame with the upset can be corrected by either overwriting the upset using the contents of the golden data, or by inverting the single bit at the location given by the ECC syndrome (see Section 4.1.2) in the readback data with the error and then writing that data back into the configuration memory. The advantage of readback scrubbing using the golden data is that all upsets can be detected because of the word-by-word comparison which reveals the exact bit locations of all upsets. The advantage of using the ECCs is that they require a much smaller storage space to hold the golden ECC values (1 word per frame instead of 101 words per frame for the 7-Series). Unfortunately, using the ECC values for correction only achieves a SECDED capability since ECCs cannot correct MBUs.

Readback scrubbers are more complex than blind scrubbers, but they do provide the user with much more information about the upsets that occur. They offer the same robustness as blind scrubbers in upset correction which includes MBU correction if using golden data.

Readback scrubbing does require a significant amount of bandwidth to issue the readback command, receive the data, compare the data to the golden data, and write from the golden data back to the configuration if an upset was detected. Such a constraint could be a limiting factor in some user designs.

5.4 Scrubbing Examples

This section will present several different scrubbing architectures that have been developed for previous generations of Xilinx FPGA families. The FPGA families considered here are the Virtex-4, Virtex-5, and Virtex-6 families. The scrubbers in this chapter were selected because they give a general sample of the different approaches and styles that are employed when creating new and improved scrubbers. Note that some scrubbers were tested on multiple generations of FPGA families, but are grouped under a single sub-header for organizational purposes.

Apart from radiation beam testing, artificial fault injection is another means of validating a given scrubbing architecture. Fault injection consists of manually reading configuration frames, intentionally inverting one or more bits, and writing the result back into configuration memory. The scrubber is typically dispatched immediately after the injection and measurements can be taken to see how much time elapses between the injection and the point where the scrubber has corrected the upset. Many of the scrubbing solutions presented in this section were validated either using radiation beam testing or fault injection.

5.4.1 Virtex-4 Scrubbers

Although the majority of Virtex-4 FPGAs are considered legacy devices, some applications developed today still target these FPGAs. The Virtex-4 devices were among the earlier FPGA families to see an increase in the development of more improved upset mitigation techniques because Xilinx supported a space grade version of the Virtex-4 FPGAs.

The NASA/GSFC Radiation Effects and Analysis Group developed a blind external scrubber that used the SelectMAP interface [13]. The publication compared the effectiveness of this scrubber to a scrubber produced by Xilinx which would later evolve into the

SEM IP (see Section 5.5.2). The NASA scrubber outperformed the Xilinx scrubber in both performance and reliability as the NASA scrubber could correct MBUs.

The Xilinx scrubber used the ICAP interface to perform the Readback-with-ECC strategy. Its main weakness was its inability to correct anything beyond SBUs. Both of these scrubbers were tested in a radiation beam on both Virtex-4 and Virtex-5 devices.

The scrubber developed in [15], used a PicoBlaze soft processor working through the ICAP to perform Readback-with-ECC scrubbing on Virtex-4 FPGAs. The PicoBlaze program instructions were stored in a BRAM that was protected with its own mitigation mechanism. The PicoBlaze scrubber could not correct MBUs, however, as discovered in radiation testing. This scrubber had a T_D of 14.6 ms.

Recognizing the inability of SECDED scrubbers to handle MBUs, [23] developed an ICAP scrubber that also used the Readback-with-ECC strategy for Virtex-4 FPGAs, but added significant enhancements. The improvements in this architecture were the use of TMR to survive isolated SBUs without breaking, and the addition of a special voter mechanism in the Digital Signal Processing (DSP) blocks. The voter mechanism was unique because it could not only alert the scrubber when an upset was detected, but also convey in which of the three modules the upset had occurred. It was also more reliable because DSP blocks are more resistant to radiation effects compared to SRAM configuration memory. The Mean Time to Detection (MTTD) was measured to be 97.93 ms. The scrubber was immediately dispatched to correct the faulty module using a Golden file stored in flash memory. As long as two of the TMR modules did not incur upsets, this scrubber could handle MBUs.

The scrubber presented in [42] was a hardware controller circuit that used the SelectMAP pins on a Virtex-4 device. The solution used a readback strategy with golden data, and thus required that the golden data be updated to reflect any new changes to the memory after a partial reconfiguration. The scrubber reads the configuration data from the SelectMAP pins while simultaneously reading the golden data stored in the DDR memory. The scrubber calculated frame-wide CRCs for each frame to detect upsets. Differences in the resulting CRCs would prompt the configuration to be overwritten with the contents of the golden data. The total scrubbing time of this scrubber was $T_s = 174.56$ ms, while $T_D = 92.05$ ms.

A fusion of TMR and Dual-Modular Redundancy (DMR) techniques was created to design a more reliable ICAP controller for the Virtex-4 [8]. Two ICAP controllers placed in close physical proximity to the ICAP combined to produce a DMR system that also included a mechanism for detecting when the DMR failed. Only one of these controllers actually controlled the ICAP, while the other was used as a comparison mechanism for error detection. A TMR recovery controller was placed in a separate reconfigurable portion of the FPGA (but still close to the ICAP) to reconfigure the DMR circuit should an error be detected. Occasionally, the ICAP controller would blind scrub the TMR circuit. Frame CRCs were used to detect upsets in the normal configuration memory. Correction was carried out using a Golden file stored in an external rad-hard flash memory. The whole idea behind this design was to reduce the amount of single points of failure in the architecture. This mechanism was validated using fault injection experiments.

5.4.2 Virtex-5 Scrubbers

Virtex-5 FPGAs are fabricated using a smaller process technology (65 nm) than previous FPGA families. The latest RHBD FPGA is a Virtex-5 FPGA [12]. Virtex-5 FPGAs are still commonly used is many applications at the present day.

The scrubber in [52] used the ICAP with an FSM as the processing logic, resulting in a lower resource utilization and faster error correction and recovery time. The detection mechanism was a Readback-with-ECC strategy, and correction came from using golden data. This architecture was implemented on both Virtex-4 and Virtex-5 FPGAs. The T_D of the Virtex-4 implementation was 1.477 ms, and the T_D for the Virtex-5 version was 2.261 ms. T_c was 2 μs for both versions. This scrubber was an attempt to improve upon the scrubber presented in [42]. The validation of this architecture took place by means of an "SEU Fault-Emulation" experiment.

The design given in [26] consisted of a soft multi-processor System-on-Chip (SoC) on the Virtex-5 device. Two Microblaze cores, essentially, scrub each other. One core enacts a Readback-with-ECC strategy until an error is detected. The processor then attempts to correct that error unless it hangs. A watchdog will then trigger the other core to correct the fault in the first core. Both cores connect into the ICAP interface via a Xilinx IP AXI

Hardware ICAP (HWICAP) module. This module enables efficient communication from the processors to the ICAP. Unfortunately, this scheme could not handle MBUs, but it did provide another example of using dual scrubbers to watch one another, a relatively novel idea at the time. This architecture was also validated in an SEU Fault-Emulation experiment.

Since the use of soft processors in FPGA designs has become increasingly more common, soft processor reliability techniques such as using checkpoints have become more important. The work in [25] developed a checkpoint and rollback scrubber to correct upsets and recover efficiently for a Leon3 system implemented on a Virtex-5 FPGA. When the soft processor was completely reconfigured, the state values may not have been synchronized, thus, restoring back to a checkpoint or some known state was a more desirable recovery. Once the error was detected, the error was scrubbed, and the state of the design was reverted back to most recent checkpoint. Their work also showed that the amount of sensitive design bits could be significantly condensed into fewer frames, and that potentially those frames were the only ones that would require any scrubbing. Detection was accomplished by checking the traditional ECC and CRC codes when reading back, with a T_D of 5.9 ms.

Apart from developing new scrubbing solutions, there has also be research attempting to increase performance gains with existing strategies. The research of condensing sensitive bits into sensitive frames and only scrubbing those frames is one such example. Other examples such as the scheme presented in [53] suggests shifting the default start frame from address zero to wherever will statistically cover the most critical bits the soonest, minimizing the Mean Time to Repair (MTTR). This technique, validated on Virtex-5 FPGAs, uses the ICAP and requires little extra circuitry. The work by [5] showed how the scrubbing task on-board an FPGA with many concurrent processes could be dynamically scheduled to more frequently scrub critical tasks to help improve overall reliability.

5.4.3 Virtex-6 Scrubbers

Virtex-6 FPGAs were fabricated with even smaller process technology (40 nm) compared to their predecessors. A multi-layered encoding was developed by [54] for Virtex-6 FPGAs that mapped each configuration frame to a 2D matrix. Subsequent Hamming and parity codes were then computed across three directions of the matrix: by row, by column,

and along the diagonal. The detection process was performed by reading back each frame using the ICAP, mapping the data to the 2D matrix, computing the Hamming and parity codes, and comparing the codes to values stored in a memory. Apart from being very complex and increasingly difficult to implement on larger FPGA devices, this scheme could handle MBUs. The experiments performed with this scheme consisted of Matlab simulations.

The work of [55] introduced a novel bit-interleaving scheme to be used in coordination with a Readback-with-ECC strategy for Virtex-6 devices. Each frame was divided into sub-frames, and each sub-frame was given an ECC code that was stored in non-essential bits of that frame. This feature allowed the readback to compute frame and sub-frame ECCs in order to detect both SBUs and MBUs, although large MBUs could still break the code. Storing the sub-frame ECCs inside of the frames themselves avoided the use of an external memory. This architecture was validated using fault injection experiments.

Another unique correction mechanism was presented by [56] for Virtex-6 FPGAs. This mechanism harnessed the power of erasure codes. The idea behind erasure codes is to use redundant blocks of memory to restore corrupted blocks from the remaining uncorrupted blocks. Exact locations of upsets were not necessary, but basic error detection was required. This work utilized erasure codes for correction and an interleaved parity scheme for detection. Interleaved parity is where a parity code at various bit-distances and in multiple dimensions is applied to the data. The purpose of these special parity bits was to correct MBUs, and the solution did so successfully, with a few isolated exceptions. The blocks of data for the erasure codes were physically spaced throughout the FPGA, and multiple frames were processed as clusters with the redundant blocks stored in BRAM. The scrubber read back frames, performed IND detection (interleaved parity with N dimensions) with a T_D of 18.7 ms, and used erasure codes to correct errors.

5.5 Hybrid Scrubbing

With the innovation of the Readback CRC mechanism, it became desirable to create a scrubber that could take advantage of the built-in Readback CRC hardware to detect and correct SBUs, which are the majority of upsets [56]. At the same time, another scrubbing technique could be used to correct MBUs, since the Readback CRC could not. The combina-

tion of these two mitigation strategies to handle all forms of upsets led to the term "hybrid" to signify the two separate techniques being used in one architecture. Hybrid scrubbing can be thought of as the combination of CRC-based scrubbing and Frame ECC-based scrubbing [9]. The built-in hardware handles all SBUs and detects MBUs; its ECC and CRC calculation results display on the outputs of the Frame ECC (see Figure 4.3). One of the key aspects of this new architecture was the total reliance of the scrubber upon the and Readback CRC and the Frame ECC primitive for error detection.

The goal of this architecture is to achieve the benefits of each of the two member scrubbing techniques that overshadow the weakness of the other. Hybrid scrubbing achieves tremendous performance gains offered by the dedicated hardware to handle SBUs, the majority of upsets. At the same time, a more robust scrubbing strategy like blind or readback is used to correct MBUs, achieving a higher level of robustness.

One of the minor limitations that early hybrid scrubbers often exhibited was sometimes employing less-than-efficient scrubbing responses. Whenever a CRC error occurred, for instance, early hybrid scrubbers would often scrub the entire device rather than just the frame with the error.

This section gives the history of hybrid scrubbers including citations to publications demonstrating their effectiveness in radiation testing. Among these hybrid scrubbers is the scrubber developed by Xilinx, the SEM IP.

5.5.1 Early Hybrid Scrubbers

Basic scrubbing strategies, like blind and readback scrubbing, had an advantage in that they could correct all types of configuration upsets. Their disadvantages were the large processing bandwidth requirements of continuously sending and receiving configuration data, significant power consumption (especially if no delays were used), and slow correction response due to the amount of time needed to traverse the entire device (in a worst-case scenario).

The primary forerunners of the hybrid architecture presented in this thesis include the scrubbers presented in [16] and [3]. This section will describe the former while Section 5.5.2 will describe the Xilinx-recommended scrubber, the SEM IP.

The scrubber in [16] was successful at correcting MBUs and was implemented on a Kintex 7-Series FPGA using the JTAG interface. T_D for this scrubber was the speed of the internal scan, 30 ms. Whenever MBUs occurred, this scrubber would scrub the entire device in about 115 seconds, making the scrubber limited from a performance perspective [57]. However, this scrubber was validated in a radiation beam correcting all SBUs and MBUs of various sizes.

5.5.2 Xilinx Soft Error Mitigation IP (SEM IP)

Xilinx created the SEM IP as an upset mitigation mechanism for terrestrial applications [3]. The SEM IP utilizes the internal Readback CRC hardware to perform error detection using the built-in ECC and CRC codes [58]. The SEM IP, however, performs its own correction (as opposed to letting the Readback CRC perform the correction) by means of the ICAP. This correction includes capabilities for MBU correction from a golden data.

The SEM IP has several levels of customization depending on the needs of the user design. It includes the ability to turn off or on many features to conserve potentially limited resources like power, area utilization, etc. One of these features is a classification component which can determine whether an upset bit was an essential bit or not. It uses the generated essential bits file discussed in Section 3.4.1 to accomplish this task. The classification of errors allows the SEM IP to avoid resorting to potentially disruptive recovery procedures for bits that will not affect proper design function.

The SEM IP also offers different modes of error correction. Repair mode is essentially the equivalent of the Readback CRC which handles all SBU correction, but cannot correct MBUs. Enhanced Repair mode adds individual frame CRCs to support SBU and double-bit adjacent error correction, but cannot correct other MBUs. Replace mode is essentially the same as blind scrubbing in that several frames are written from the contents of golden data stored in a large memory. Replace mode can correct all larger MBUs.

Detection for the SEM IP is accomplished by means of the Readback CRC. The detection speed depends on the number of frames of the FPGA and the clock frequency of the Readback CRC. For instance, T_D for the XZC07020 FPGA with a Readback CRC frequency of 100 MHz is 8.02 ms, which equates to a T_d of 1 μs. The SEM IP's correction

speeds are given in [3] on page 20. T_c for SBUs is 610 μs using Repair or Enhanced Repair correction modes. Two-bit adjacent errors have a T_c of 18.79 ms when using the Enhanced Repair mode. The Replace correction mode corrects all errors from golden data, thus all error types have a T_c of 830 μs.

While the SEM IP circuit is fairly complex (see Figure 5.4), it is also relatively small in its resource utilization (see Table 7.3 for a breakdown of SEM IP resource utilization on a Zynq FPGA). All of these internal components are, however, just as susceptible to the same radiation effects as the rest of the FPGA.

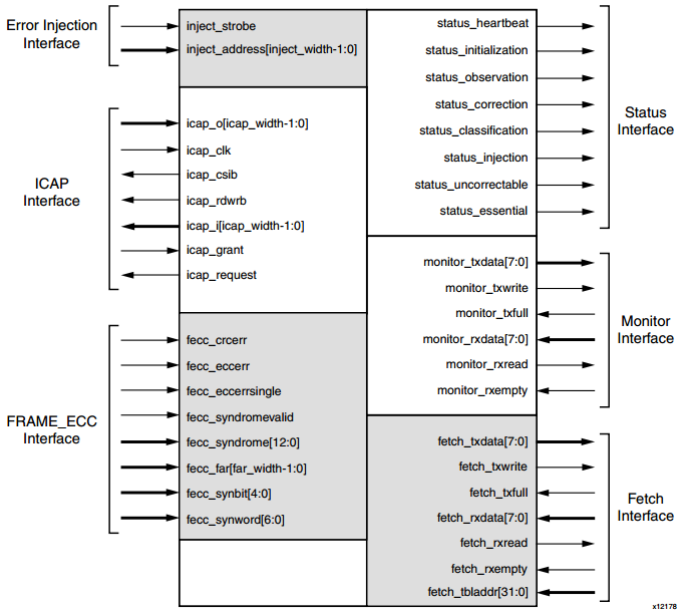

Figure 5.4: SEM IP Controller Ports [3] pg. 27

If a CRC-Only error is detected (referring to a CRC error with no ECC information), the SEM IP considers this error "uncorrectable" and requires a full reconfiguration to return to proper functionality. The SEM IP also does not support any partial reconfiguration performed by the user [59]. Despite these minor drawbacks, it is one of the fastest architectures for detection and SBU correction, and is superior in many ways to other scrubbing mechanisms.

In order to create new and improved scrubbing architectures, it is useful to examine what has already been developed. This chapter presented an overview of basic scrubbing techniques and gave examples of several scrubbers that have been developed previously. The background presented in this chapter will serve as the foundation for the scrubbing architectures introduced in the next chapters of this thesis.

Chapter 6

7-Series Scrubbing Architectures

The purpose of this chapter is to present scrubbing architectures built for Xilinx 7-Series devices. This chapter introduces blind, readback, and hybrid scrubbing architectures. All of these architectures target 7-Series FPGAs as opposed to earlier generations, which already have many scrubbing architectures available (see Chapter 5).

The first section highlights important issues regarding frame boundaries that must be understood by 7-Series scrubbers. The next section gives a brief summary of the necessary compilation files used by these scrubbers. The subsequent sections introduce blind, readback, and hybrid scrubbing architectures for the 7-Series FPGAs. This chapter will describe how the scrubbers work, while Chapter 7 will demonstrate how two of them are implemented on an actual FPGA.

6.1 Frame Row Boundaries

Chapter 4 described the physical organization of frames tiled into rows and columns throughout the FPGA. There are two unusual behaviors that scrubbers encounter when reaching frame row boundaries during reads and writes to the configuration memory. These behaviors can cause problems when scrubbing if not understood correctly. The first concerns the auto-incrementing behavior of the Frame Address Register (FAR) when the row boundaries are reached. The second deals with accurately writing or reading data around the row boundaries. The solutions to these issues are related, and are thus presented together in this section.

6.1.1 FAR Auto-Increment Plateauing Behavior

When reading or writing frames, the FAR auto-increments when it detects that an entire frame has been processed. Before a readback of multiple frames is requested, the FAR is written with the address of the first frame to be read. As each frame passes through the configuration module, the FAR increments to the next frame address automatically (see Section 3.3.3 under **FAR**). Thus, if address `0xA0` is specified as the first frame, the FAR holds the address `0xA6` after a readback of six frames has occurred.

Configuration operations can become misaligned due to the auto-increment feature of the FAR. One of the idiosyncrasies associated with the auto-increment feature is that it stops incrementing when a row boundary is encountered. The event is most commonly observed when reading back only **one** frame at a time, and the row boundary is reached. If one frame is continuously requested, the FAR will never increment past the last address of the current row. The row that a frame resides in is indicated by bits [21:17] of its FRAD (see Figure 3.2). Whenever two sequential FRADs have different row numbers (as indicated by the row bits in their FRADs), the transition is considered a row boundary (see Table 6.1).

To address this behavior when performing readback, multiple frames must be read back in one transaction when a row boundary is reached. The number of frames to be read back has to be at least one more than the distance to the boundary. For example, as shown in Table 6.1, a given FRAD sequence is `0x2A48`, `0x2A49`, `0x40000`, `0x40001`, `0x40002`, etc. The transition from `0x2A49` to `0x40000` crosses a row boundary. If the current value in the FAR is `0x2A48`, then a single transaction requesting at least three frames (no fewer) must be requested for readback to increment the FAR past the boundary (the distance to the boundary from the current frame (`0x2A48`) is two frames). The same solution is true when writing to the configuration.

Table 6.1: FRAD Row Boundary

Row A	...	...	...	...	0x2A48	0x2A49
Row B	0x40000	0x40001	0x40002	...	...	...

6.1.2 Using Dummy Frames to Overcome Row Boundaries

Row boundaries also create problems for accurately writing or reading the configuration data. When a row boundary is encountered during a write to the configuration, a frame's worth of data will be lost, and subsequent frame data will be misaligned (i.e. subsequent frames will not match the frame locations that they were intended for). Using the same example in the previous section, if the FAR holds the value `0x2A48` and the user wants to write the next five consecutive frames, the user has to append two extra "dummy" frames at the appropriate position of the write sequence to account for the row boundary. The resulting write sequence is shown in Figure 6.1.

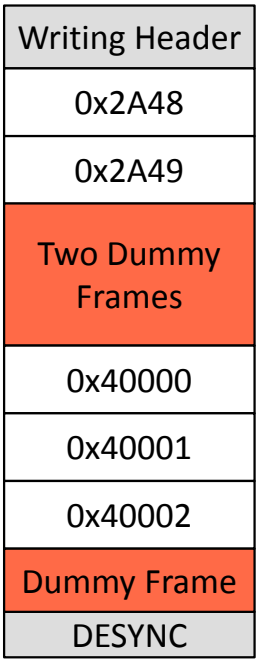

Figure 6.1: Write Configuration Sequence to Overcome Row Boundary

The sequence would consist of the data for addresses `0x2A48` and `0x2A49`; two dummy frames for the boundary; the data for addresses `0x40000`, `0x40001` and `0x40002`; and a final dummy frame that accounts for the frame buffer (see Section 3.4.2). The total size of the entire write sequence would be the number of words for the header and DESYNC commands plus 808 words (assuming a frame size of 101 words) of data and dummy frames (202

(0x2A48, 0x2A49) + 202 (boundary) + 303 (0x40000 to 0x40002) + 101 (frame buffer) = 808). The dummy frames lift the auto-increment over the boundary and allow the configuration memory write to continue normally. Properly formatted bitstreams by the vendor tools automatically handle this issue by including pairs of dummy frames at each row boundary.

A similar manifestation of the boundary issue is also encountered when performing readback. Recall from Section 3.4.2 that one extra frame must always be requested in a readback along with the desired amount of frames to account for the frame buffer. When a row boundary is encountered, an additional two dummy frames must be requested to overcome the boundary. If the FAR initially holds the FRAD 0x2A48 and 606 words (6 frames) are requested, then the data from 0x2A48, 0x2A49, and 0x40000 is read while the other three frames are absorbed by the frame buffer and the compensation of the boundary jump. Interestingly enough, the FAR now holds the value 0x40002 (see Table 6.1). If further readbacks are performed without first writing a new FRAD to the FAR, the data from 0x40001 is skipped entirely because the FAR auto-incremented passed that FRAD already and now holds the value 0x40002.

6.2 Scrubbing Files

Before commencing the discussion on scrubbing architectures, it is necessary to discuss some of the files that the scrubbers of this thesis require for compilation or for use during runtime. Most of these files are altered versions of files generated by Xilinx tools. These files are necessary for the scrubbers of this thesis to perform properly and also assist in debugging.

6.2.1 Golden Data

The golden data is the original configuration data that should always be present in the FPGA and is typically stored in an off-chip memory (i.e. DDR RAM, external SPI flash [3], etc.). It is a somewhat cumbersome task to obtain this golden data, however. There are two methods presented here for obtaining this data.

The first method is to extract the golden frame data from the original bitstream. Simply using the bitstream itself as the golden data is not suitable because it contains

header and footer data as well as Type 1 BRAM frames which are not scrubbed (see Section 3.2.2). The solution is to take the original bitstream and strip out the non-configuration data (i.e. command sequences), leaving behind only frame data. It is useful, however, to leave the dummy frames found at row boundaries inside the golden data to facilitate single-transaction readbacks of more than one frame.

An alternative means of obtaining the golden data is achieved by performing an initial readback of the entire device immediately after configuration. In this method, the process of extracting the frame data out of the bitstream is avoided. The initial readback needs to be performed as quickly as possible after the initial configuration process so as to avoid incurring upsets that will be consequently saved into the golden data.

Furthermore, if partial reconfiguration is performed (see Section 3.5), then the scrubber can simply perform another full readback afterwards which then can replace the old golden data. The new golden data automatically includes the newly programmed areas of the FPGA.

In order to use this initial readback option, the GLUTMASK signal must be enabled (see Sections 3.4.1 and 6.2.2 for more information). Any masked data will then be masked on both the initial and subsequent readbacks.

6.2.2 Mask File

Recall from Section 3.4.1 that some configuration bits change with normal design behavior [1]. The mask (MSK) file indicates the location of these masked bits in the configuration memory. Without masking, these bits would indicate errors when compared to the golden data, since the golden memory contains the initially programmed values. Each "1" in the MSK file maps to a "don't care" bit in the bitstream. Thus, an error can be detected using the readback strategy by applying the following computation:

$$(g_i \oplus r_i) \wedge \neg m_i, \tag{6.1}$$

where g_i, r_i, and m_i are the i^{th} golden, readback, and masked bits respectively.

The built-in Global LUT mask (GLUTMASK) signal (introduced in Section 3.3.3 under **CTL0**), replaces the need to use a MSK file. When enabled, the GLUTMASK signal automatically masks any dynamic bits on all readback data. The masked readback data is read back as all zeros or all ones [3], even though the actual values of the bits may be changing. The GLUTMASK signal is the preferred means of locating masked bits as it does not require an external memory to store the MSK file.

It should be noted that the GLUTMASK signal can only be used in tandem with the initial readback method of obtaining the golden data and not with the edited bitstream scheme. An edited bitstream scheme contains the initially programmed values of masked data. If the GLUTMASK signal is enabled for subsequent readbacks, those results will be masked and will differ from what the edited bitstream contains.

6.2.3 Essential Bits File

Similar to MSK file generation, the bitstream tools can also generate an *Essential Bits* file (see Section 3.4.1). Essential bits are bits which are known to control the circuitry of the user design [44]. Thus, an upset to an essential bit could potentially disrupt the functionality of the user design. This file is optional to scrubbing, as it merely grants scrubbers the opportunity to report on the "severity" of an upset bit. The SEM IP is built to optionally use this file [3].

6.2.4 Frame Address (FRADs) List

A list of all of the FRADs for the target device in proper order is necessary for the scrubbers presented in this chapter. While this requirement is not true for all scrubbing architectures, the scrubbers of this thesis rely upon the FRADs list to ensure that scrubbing operations are performed on the correct frames (i.e. by first writing the FAR with the FRAD of the frame to operated upon). Other scrubbing architectures may generate this list on the fly via dedicated circuitry, or may not use a FRADs list at all.

Knowing the sequence of FRADs is also useful for debugging purposes. For instance, performing fault injection into a specific frame for testing purposes requires that the valid address of the desired frame be first written to the FAR. After injecting an upset into a given

frame, the FAR auto-increments. The FRAD corresponding to the now-upset frame must be rewritten to the FAR to verify that the injection was successful by reading that frame back. Correcting a fault also requires that the FAR be written with the specific FRAD where the upset was detected followed by writing valid configuration data.

As an example, Appendix F gives excerpts from a FRADs list for a Zynq SoC device. A list of FRADs can be obtained by one of two means:

1. **DEBUG_BITSTREAM**

   A special bitstream option[1] will generate a "debug bitstream." This bitstream writes each frame individually instead of starting at the very first frame and writing to the whole configuration at once (which normal bitstreams do). Thus, each FRAD is explicitly declared in the bitstream. These FRADs can easily be obtained by parsing the debug bitstream and compiling a list using scripting programs. However, since the FRADs come from the bitstream, they typically consist of only Type 0 and Type 1 FRADs (see Section 4.1.1). Furthermore, this feature is not supported for all device families.

2. **Readback Frames**

   Since reading back frames auto-increments the FAR, this auto-incrementing feature can also be exploited by traversing across the entire device to obtain all of the FRADs. As shown in Figure 6.2, the reading procedure starts by writing the first FRAD (always 0) to the FAR. Then, exactly one frame is repeatedly read back, pausing only to read the FAR in between each readback. A condition must also be checked to determine whether the FAR is plateauing (i.e the most recently read value of the FAR is the same as the value read previously). If so, then a row boundary has been reached and two frames must be read back to overcome the plateauing behavior of the FAR's auto-increment feature (refer to Section 6.1.1). The readback stops when Type 4 frames are reached. In addition, Type 1 frames need to be removed from the list as they are not scrubbed either. The disadvantage of this approach is that the actual FPGA is needed

[1]The BitGen option for creating the Debug Bitstream: -g DebugBitstream.

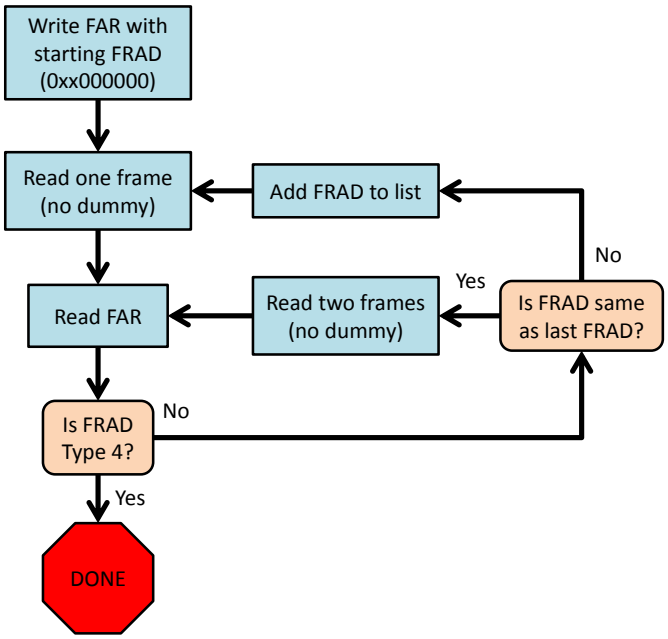

Figure 6.2: FRADs List Readback Frames Method

to perform frame readback as opposed to just using software tools to generate the list as in the Debug Bitstream case.

6.3 Blind Scrubbing Architecture

The most basic of the three scrubbing architectures is known as blind scrubbing (see Section 5.3.1). The scrubbing is "blind" because no upset detection mechanism (i.e. readback, ECC, CRC, etc.) is used. Instead, the configuration is continuously overwritten with the contents of the golden data. Blind scrubbing is typically performed by writing the entire configuration at once (full blind scrub) or by writing one or more frames at periodic intervals. Figure 6.3 shows the basic control flow of the blind scrubbing strategy.

The blind scrubber starts at the first frame (typically `0x0000000`) and writes one or more frames from the golden data into the configuration. If the scrubbing is broken up into one or more frames, each write command sequence must be recreated with the current frame's golden contents before being sent to the configuration module. The sequence is assembled by starting with the write header commands, extracting and appending the next

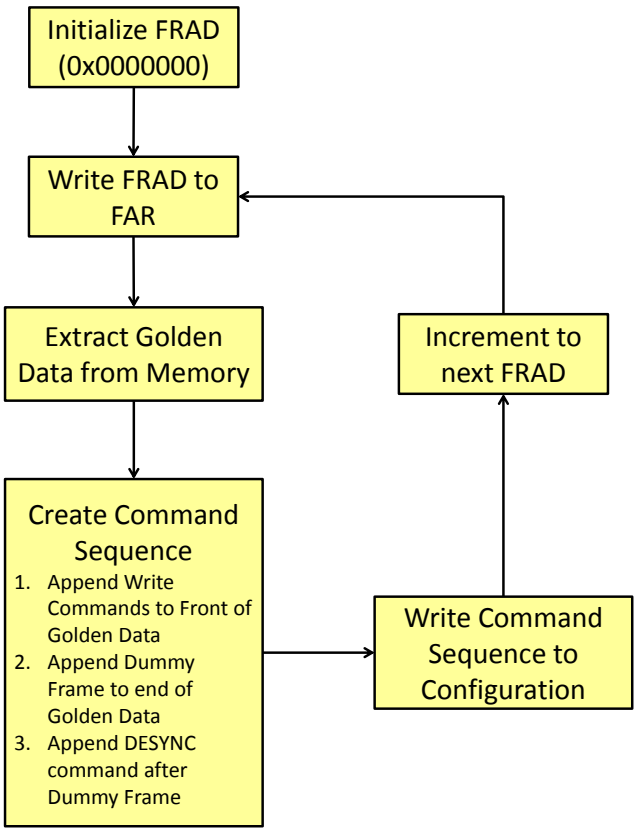

Figure 6.3: Blind Scrubbing Flow Diagram

golden data contents, and adding the DESYNC command to the end (see Appendix D for examples of how to create these command sequences).

Unless the scrubber is performing a full blind scrub, the scrubber must write the FAR with the next FRAD according to the FRADs list between each write transaction. Care must be taken to ensure that the row boundaries are crossed accurately (see Section 6.1.2). When the end of the device is reached, the next FRAD loops back to the starting address.

As explained in Section 5.3.1, the most significant benefits of blind scrubbing are the correction speed and the simple design of the architecture. It also corrects all configuration upsets. The primary drawback to using this architecture is that no upset detection occurs. Therefore, it is not very interesting since it cannot report on any of the upsets that it is correcting, and it is difficult to measure its effectiveness.

6.4 Readback Scrubbing Architecture

The readback scrubbing architecture introduced in this thesis uses a Readback with golden data strategy (see Section 5.3.2). Originally, this architecture was expensive from a memory perspective since it required both a golden and mask file to be stored in on-chip memory. The "initial readback" procedure (discussed in Section 6.2.1) was developed to simplify the creation of the golden data. The initial readback reads back the contents of the entire configuration memory and stores the data. This procedure, along with enabling the GLUTMASK feature, eliminated the need for a MSK file or an edited bitstream.

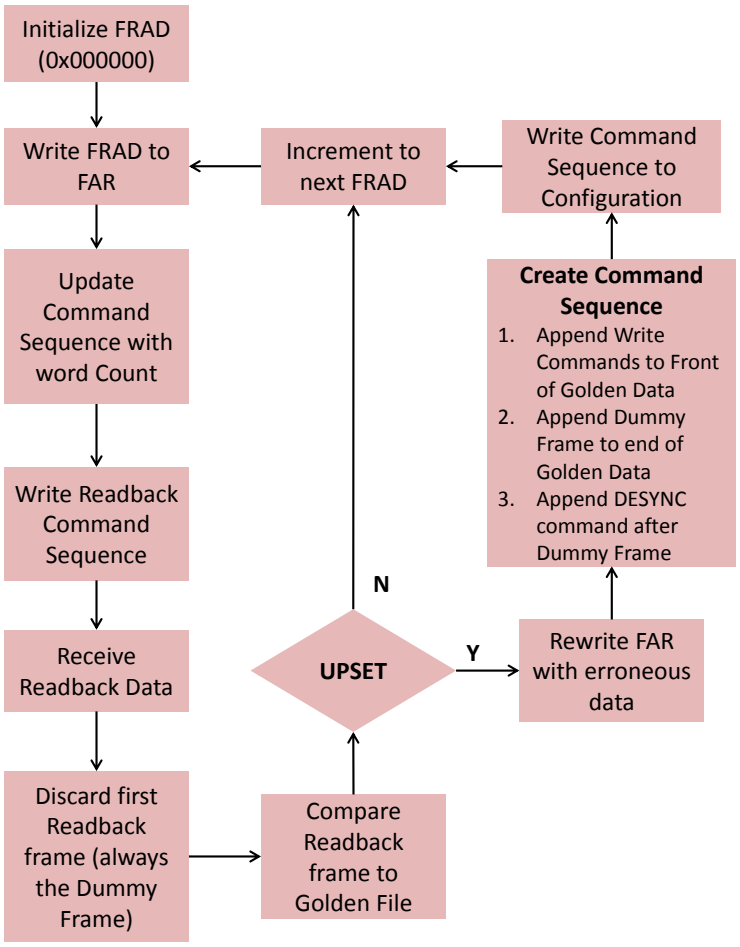

Figure 6.4: Readback Scrubbing Flow Diagram using Golden Data

The basic readback flow diagram is shown in Figure 6.4. The readback scrubbing process writes to the FAR with the first FRAD, then issues the readback command sequence to read one or more configuration frames. Table D.6 in Appendix D shows a readback sequence. After receiving the readback data, the first frame is discarded since it is a dummy frame (refer to Section 3.4.2). The scrubber then compares the data, word by word, to the corresponding data in the golden data (stored in an external flash or DDR memory). If a difference exists, then the scrubber generates an error message reporting the exact frame, word, and bit locations (see Appendix G for an example log output). Subsequently, the FAR is rewritten with the FRAD whose frame contains the erroneous data and the scrubber assembles a correction write command sequence. The affected frame in configuration memory is then overwritten with the corresponding golden frame data. Note that a complete FRADs list is needed to perform these actions.

Performing readbacks continuously can be a resource-taxing process for some systems in terms of memory bandwidth. It may be preferable to insert delays in between readbacks, or to read fewer frames at a time if bandwidth is limited. However, unlike blind scrubbing, readback scrubbing provides error detection while maintaining a fairly simple architecture. It has slower correction performance compared to blind scrubbing due to the time taken in performing readback before correcting upsets.

6.5 Hybrid Scrubbing Architecture

This thesis introduces a new hybrid scrubbing architecture in an attempt to create a scrubber that lacks the drawbacks of the previous two architectures yet maintains their advantages of performance and robustness. Inspired by the hybrid scrubbers described in Section 5.5, this section presents the new hybrid scrubber that handles all of the special upset types, improves the efficiency of scrubbing techniques, and achieves correction speeds close to those of the Xilinx SEM IP [3]. This section presents an architecture that implements these goals.

As shown in Table 6.2, the hybrid scrubber harnesses the performance benefits of the Readback CRC mechanism to correct SBUs and detect all upsets. Where the Readback CRC cannot correct MBUs, an external scrubber can step in to correct these larger upsets.

This hybrid architecture utilizes a readback scrubber to perform this task of correcting larger MBUs. Hence, the robustness of correcting all upsets is also preserved in addition to capitalizing on the performance advantages of the Readback CRC.

Table 6.2: Hybrid Scrubbing Detection and Correction

Type of Upset	DETECTION	CORRECTION
SBUs	Readback CRC	Readback CRC
MBUs	Readback CRC	External Readback/Blind Scrubber

The 7-Series CRC code, which the Readback CRC mechanism uses, can detect up to a 31-bit upset in a single frame with 100 percent accuracy [47] (see Section 2.2.3), and the accuracy is near 100 percent for larger upsets [49]. SEUs causing multi-bit upsets of this magnitude are not realistic, and the scrub rate is assumed to be much greater than the upset rate to prevent an accumulation of upsets past 31 in a single frame. Virtually no upsets can break both the ECC and CRC codes and go undetected.

The hybrid scrubbing architecture of this thesis will be presented in the following sequence: first, an overview of the physical structure of the hybrid scrubber will be given. Second, a description of the control flow of the hybrid scrubbing logic will be introduced. The implementation of this architecture will be discussed in Chapter 7.

6.5.1 Hybrid Scrubbing Components

The hybrid scrubbing architecture introduced in this work consists of the following elements: the built-in Readback CRC mechanism, the `FRAME_ECCE2` primitive, a buffer such as a FIFO to store `FRAME_ECCE2` data, and the scrubbing logic executing the scrubbing algorithm (see Figure 6.5 for a diagram). Note that most of the elements have to be implemented in hardware (typically inside the FPGA itself). The exception is the scrubbing logic which may be implemented in either hardware or software.

Scrubbing logic implemented in software is outside of the FPGA either because it is running on a built-in processing system (PS) situated right next to the programmable logic (as in the case of the Zynq SoC), or on another micro-controller completely external to the

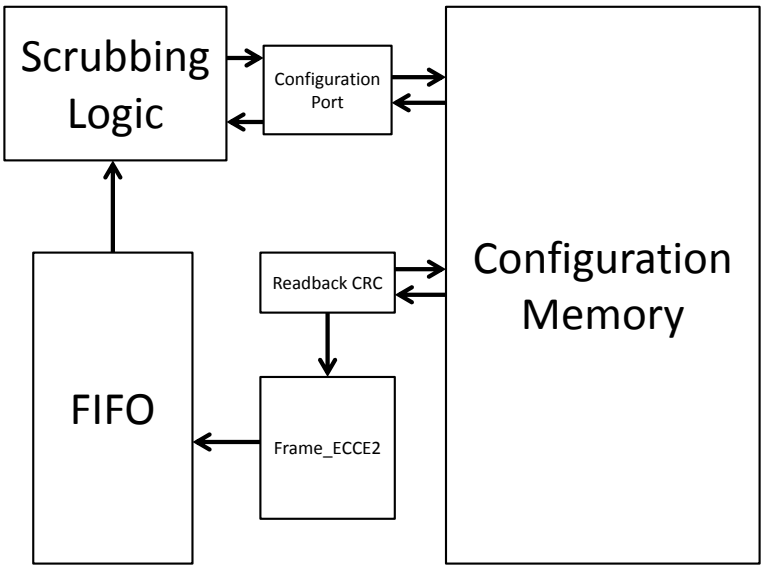

Figure 6.5: Hybrid Scrubbing Architecture Components

target FPGA. Scrubbing logic implemented in hardware can be inside the same FPGA that it is scrubbing, or on an external FPGA.

The Readback CRC mechanism is enabled and set to the CORRECT_AND_CONTINUE mode (see Section 4.3.1). The results of the continuous readbacks performed by the Readback CRC will appear on the outputs of the `FRAME_ECCE2` primitive. Capturing a group of the synchronous `FRAME_ECCE2` outputs creates a "batch" of signals that is then loaded into a buffer (typically a FIFO). Thus, each FIFO entry is a batch of `FRAME_ECCE2` signals (i.e. all of the `FRAME_ECCE2` signals concatenated together). The FIFO passes each batch of `FRAME_ECCE2` signals to the scrubbing logic. The scrubber interprets these signals to get an accurate picture of the upset events and then quickly determines and engages in an effective repair approach. If there are no nonzero batches present in the FIFO, then the scrubber simply polls (i.e. continuously reads the FIFO's output) until nonzero batches appear. This scheme could be converted into an interrupt-driven architecture as well.

The Readback CRC behaviors presented in Section 4.3.4 result in distinct, yet predictable, quantities of `FRAME_ECCE2` batches being loaded into the FIFO. These quantities are shown in Table 6.3. A batch of signals is loaded into the FIFO when the `FRAME_ECCE2`'s SYNDROMEVALID signal is asserted and the SYNDROME value is nonzero. Odd-numbered

in-bounds upsets, (including SBUs), always generate exactly two FRAME_ECCE2 batches. The first occurs when the Readback CRC detects the upset, and the second occurs when the Readback CRC reads the frame a second time in its attempt to correct the upset (see Section 4.3.5 for a more illustrative example).

Table 6.3: Number of FRAME_ECCE2 Batches Generated by Upset Type

Type of Upset	FRAME_ECCE2 Batches Generated
SBU	2
Even-Numbered MBU	∞
Odd-Numbered In-Bounds MBU	2
Odd-Numbered Out-of-Bounds MBU	∞
Odd MBU Correction at Masked Bit	∞

Odd-numbered out-of-bounds and masked bit upsets continually generate FRAME_ECCE2 batches. Since these upsets are odd, the Readback CRC always attempts to correct the upset. However, following the attempted correction, the Readback CRC restarts back at the beginning (see Section 4.3.4 for a discussion regarding the behaviors of the Readback CRC). It repeatedly detects the same upset, fails to correct it, and then restarts back at the first address.

Even-numbered MBUs also generate a continuous number of FRAME_ECCE2 batches. The Readback CRC recognizes that the upset is an even-numbered upset and that it cannot correct the upset bits. It simply continues scanning subsequent frames until it reaches the end of the device. It then restarts, scans until it encounters the same upset, continues, etc. Each pass of the Readback CRC scan generates an additional FRAME_ECCE2 batch.

In the cases of upset types which generate a continuous amount of FRAME_ECCE2 batches, the FIFO will overflow if these batches are not emptied in a timely manner. An alternative method to preventing the FIFO from overflowing is to use logic that can identify when several duplicate batches have occurred. The logic can prevent the FIFO from being loaded with subsequent batches if they are the same as those already loaded into it. After

the batches are loaded into the FIFO, the scrubber reads these batches from the FIFO and makes decisions based on the values in the batches.

6.5.2 Hybrid Flow

The hybrid scrubbing control flow is summarized in Figure 6.6. All of the decisions in the algorithm come from what the values of the `FRAME_ECCE2` batches contain. More in-depth explanations of each of the steps are given with each number label in the diagram corresponding to the same number in the list below.

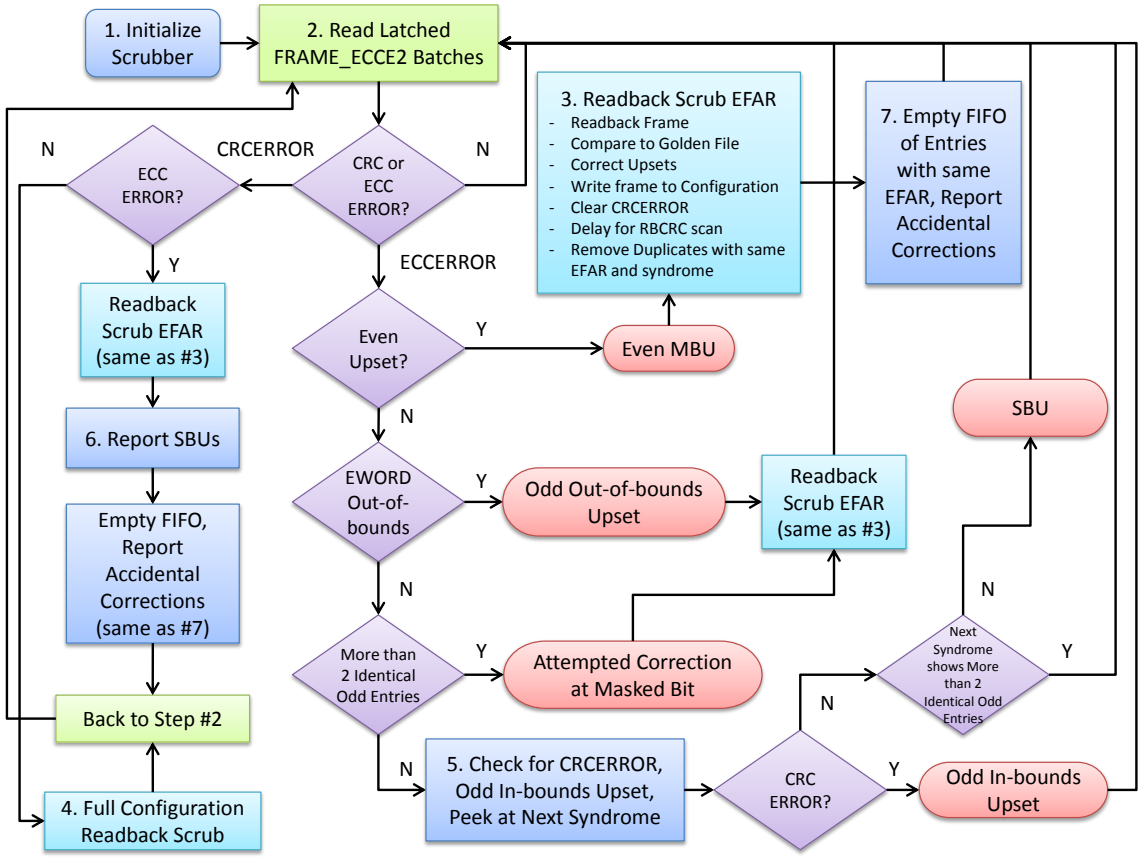

Figure 6.6: Hybrid Strategy Flow Diagram

1. **Initialize Scrubber**

   The important details of the initialization process include performing the initial readback to create the golden data (unless an edited bitstream is used; see Section 6.2.1),

writing the COR1 with the proper bits to turn on the Readback CRC (see Section 4.3.1), and enabling CRC latching (see Section 4.2.2).

2. **Read Latched FRAME_ECCE2 Batches**

   The scrubber continuously polls the FIFO's data_out signal to see if a nonzero entry is present. This task is done without continuously asserting the FIFO's read enable signal, which is only possible if the FIFO is in First Word Fall Through (FWFT) mode. This mode means that the first entry in the FIFO will always be present on the FIFO's data_out pins even though the FIFO's read enable signal has not yet been pulsed. During every polling loop iteration, global variables corresponding to each of the `FRAME_ECCE2` signals (refer to Section 4.2.1) are updated with the contents of the FIFO. If all-zero values are read from the FIFO, then these variables are updated with all zeros. When a nonzero value is read from the FIFO, the polling loop condition breaks. The next step is determined by the values present in each of the `FRAME_ECCE2` global variables.

3. **Readback Scrub the FRAD in the EFAR**

   This step can be reached under multiple conditions, but the most likely is that the scrubber has determined that an MBU has occurred in the frame specified by the EFAR signal of the `FRAME_ECCE2` (see Section 4.2.1). Whenever this step is reached, the frame is scrubbed using readback scrubbing. Then the CRCERROR signal is cleared and the Readback CRC scan is started again to see if more upsets exist. Now that the frame has been scrubbed, it is free from errors. However, the multiple FIFO entries caused by the upset type need to be cleared to avoid reporting the same error multiple times. The scrubber empties the FIFO of all such entries that have the same EFAR and SYNDROME as the MBU just corrected. Errors with the same EFAR but different syndromes indicate incorrect repairs by the Readback CRC (see Section 4.3.4 under behavior #5), which are reported separately.

4. **Full Configuration Readback Scrub**

   This step is reached when the CRCERROR signal is asserted, but the rest of the `FRAME_ECCE2` signals in the batch are zero. This type of error is known in this thesis

as a CRC-Only error. This condition means that either the MBU perfectly satisfied the ECC (so as to go undetected), or, more likely, was misinterpreted as the wrong upset type earlier and the entry was already popped off of the FIFO. This step is rarely reached and typically only occurs with the Odd-Numbered MBU → Odd-Numbered MBU/SBU upset type (which does occur). This step also exists as a fail safe to scrub the entire device in the event that any CRC-Only error occurs and no other information is known about it.

5. **Check for CRCERROR, Odd-Numbered In-Bounds Upset, Peek at Next Syndrome**

   This step is reached when the upset detected is odd-numbered, but CRCERROR has not yet been asserted. When an odd-numbered upset is detected, it is unknown whether it is a true SBU or an odd-numbered in-bounds MBU aliasing as an SBU. The CRCERROR is the signal used to distinguish the two types. Once the first odd-numbered entry has been detected, the Readback CRC is again re-enabled, and the scrubber waits for a delay long enough to give the Readback CRC time to reach the end of the device. Upon reaching the end of the device, if CRCERROR is asserted, then the error is an odd-numbered in-bounds MBU. On the other hand, if CRCERROR has not been asserted, then the odd-numbered upset is assumed to be an SBU and the Readback CRC had already corrected it.

   The scrubber must then peek at the next FIFO entry without popping it off of the FIFO. If the next syndrome indicates an odd-numbered upset which is the same as the previous two entries that were popped off of the FIFO, then an odd-numbered MBU whose ECC points to a masked bit location has been detected. SBUs and odd-numbered in-bounds MBUs only create two entries in the FIFO, so observing more than two entries means that an upset under either the odd-numbered out-of-bounds or masked bit upset categories has occurred (see Section 4.3.2).

6. **Report SBUs**

   This step is reached in the event of an SBU occurring simultaneously with an odd-numbered in-bounds or even-numbered MBU. There is a chance that the scrubber

will attribute the resulting CRCERROR to the SBU, even though the CRCERROR actually belongs to the MBU. In this case, the EFAR indicates the SBU's location causing the already-corrected frame to be scrubbed with no upsets found. The scrubber recognizes this situation and accurately reports the SBU as an SBU, and then handles the actual MBU after another iteration of reading the `FRAME_ECCE2` signals. Appendix G gives an example of this scenario.

7. **Empty the FIFO of Entries with the same EFAR**

   This step is reached whenever any MBUs have recently been detected. Unlike the action described in step #3 which removes FIFO entries having the same SYNDROME and EFAR from the FIFO, if the scrubber reaches this step, it attempts to remove all of the entries with the same EFAR but different SYNDROMEs. Since entries of the same EFAR are always consecutive, an all-zero entry (indicating no more FIFO entries) or an entry with a different EFAR always indicates that no further entries with the original EFAR will be found afterwards (unless a future upset occurs in that same frame). Thus, each entry with the same EFAR but a different syndrome indicates that either an incorrect repair has taken place, or that multiple even-numbered MBUs exist. A SYNDROME indicating an odd-numbered upset causes the Readback CRC to attempt to correct the upset. If the upset is not an SBU, this attempted correction causes another upset to be injected into the frame whose address is currently latched in the EFAR. Each of these incorrect repairs is reported, though not counted as true radiation upsets.

This chapter presented the 7-Series scrubbing architectures developed by this thesis. The inner-workings of these architectures are transparent to users to adapt to the needs of their system. The next section will demonstrate how two of these architectures are implemented on an actual FPGA.

Chapter 7

Zynq-7000 Scrubbing

This chapter will examine the Zynq implementations of two of the 7-Series scrubbing architectures that were introduced in Chapter 6. This chapter will report the limitations, challenges, performance metrics, and operation of these scrubbers in radiation beam tests. While these architectures could be implemented on any 7-Series FPGA, this chapter summarizes the Zynq-7000 implementations.

The Zynq-7000 All Programmable System-on-Chip (SoC) features a dual-core ARM Cortex-A9 processing system (PS) alongside a fabric of programmable configuration logic (PL) [43]. The Zynq SoCs are ideal candidates to run applications that potentially have both software and hardware design components. Instead of requiring an external FPGA or complex internal hardware design to implement the scrubbing logic in the configuration memory, the scrubbing algorithm can now be implemented almost entirely in software. The software runs on the ARM cores while the corresponding hardware elements are simultaneously implemented in the PL. Such a combination offers new opportunities for configuration scrubbing. The processors perform scrubbing operations using a new configuration interface known as the Processor Configuration Access Port (PCAP).

This chapter is organized as follows: first, a discussion about the unique features available when scrubbing with the Zynq Family using the PCAP will be given. This discussion will include the read and write data flows between the PS and the PL as well as important limitations to be aware of when scrubbing with a Zynq SoC. These sections will be followed by an implementation description and radiation test results of the Zynq Readback Scrubber. This chapter will conclude with the presentation of the Zynq Hybrid Scrubber along with an overview of its implementation details and radiation test results.

7.1 Scrubbing with the Zynq-7000 Family

Developing a scrubber to run in software on the ARM cores of the Zynq SoC provides a much easier environment for debugging, portability, and compilation time compared to synthesizing and generating hardware circuits in HDL. Because the Zynq has built-in processor cores, it supports the ability to host an operating system (OS) that can run the scrubbing software with other applications simultaneously [43]. When the scrubber is a software program, it is also much easier to debug, as each instruction or line of code can be stepped through using the Xilinx SDK or other debugging tool as opposed to re-synthesizing HDL or using hardware simulators.

A disadvantage of using Zynq SoCs is the susceptibility of the PS to SEUs. Just as an internal scrubber implemented in configuration memory is vulnerable to radiation upsets, the ARM processors are also exposed to the same radiation effects that can disrupt proper processor function. Hardware reliability techniques do exist for the PS such as cache parity, system-/core-level watchdogs, and ECC in the DDR memory to help mitigate upsets to the PS [4]. However, further discussion of processor reliability techniques is beyond the scope of this thesis.

This section outlines the configuration interfaces that allow software scrubbers running on the ARM processors to access the configuration memory. This summary also includes important considerations and limitations in working with these interfaces.

7.1.1 Processor Configuration Access Port (PCAP)

The PS contains a new configuration interface known as the Processor Configuration Access Port (PCAP). The PCAP is the gateway for the PS to access the PL and includes a Direct Memory Access (DMA) controller, an AXI bus interface to communicate on the PS AXI interconnect, and a pair of FIFOs (transmit and receive) (see Figure 7.1). This interface essentially grants the PS easy access to perform configuration operations (like programming a bitstream) to the PL.

The PCAP is somewhat unique among configuration interfaces in that it does not require a specialized cable or dedicated I/O pins (unlike JTAG or SelectMAP). Instead, the PCAP is accessible to the user through software by using dedicated memory-mapped

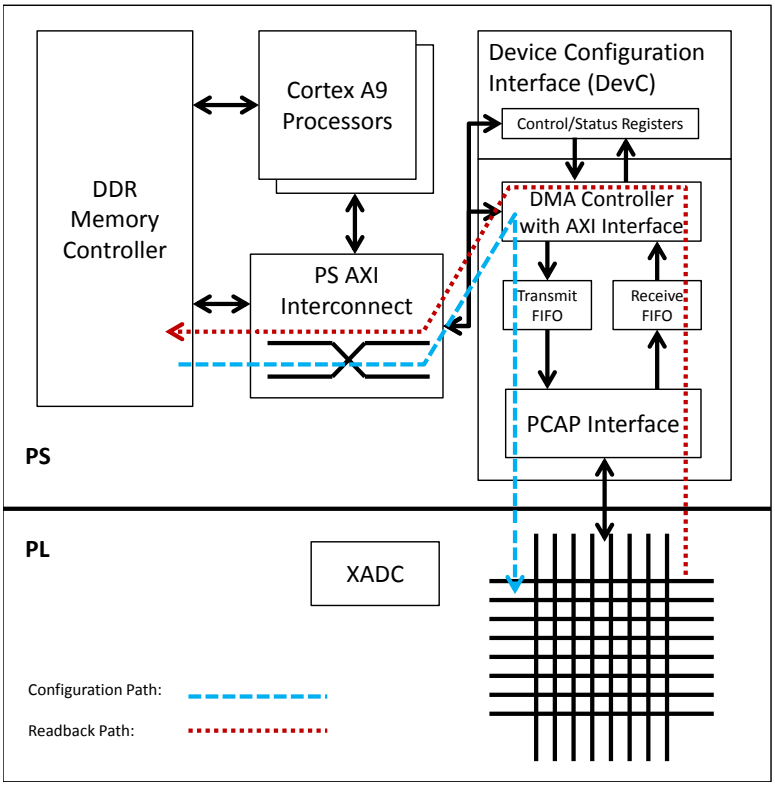

Figure 7.1: PCAP Interface and Data Transfer Paths

registers. The PCAP is an internal configuration interface (see Section 5.2), but it is not actually inside the configuration memory. A scrubber using the PCAP does not consume additional logic resources inside the user design like an ICAP scrubber requires. The PCAP scrubber runs on the dual-core ARM processors with the program's memory storage located in DDR or the caches (if enabled).

The PCAP interacts with the PL by transferring configuration command sequences (received via DMA) directly to the configuration module which processes the command packets accordingly (see the data path lines in Figure 7.1). The PCAP DMA controller acts as a master on the AXI-bus interface; the controller transfers blocks of data (such as command sequences) from DDR or other large memory location to the transmit FIFO inside the PCAP interface. The PCAP then empties the transmit FIFO and sends the commands one by one to the configuration module. If the configuration commands request a readback of configuration data, the receiver FIFO will be populated by the PCAP interface receiving

the readback data from the configuration module. The DMA controller then moves the data from the receiver FIFO into a destination memory.

One important note when using the PCAP is that the PCAP must have exclusive configuration module access. With the many interfaces that are available on the Zynq family (including the ICAP, JTAG, and PCAP), only one can access the configuration logic at a time. For this reason, certain bits in the DevC registers are used to enable[1] the PCAP interface and disable the ICAP interface.

These interfaces also compete with the Readback CRC (critical to hybrid scrubbing), which has the lowest priority among the configuration interfaces (see Section 4.3.1). The Readback CRC only runs when no other interface is running and when the last interface to access the configuration has properly sent the DESYNC command (see Section 3.3.4) [1]. Thus, for a hybrid scrubbing architecture implemented on a Zynq SoC, the PCAP interface would have to release the configuration logic anytime that the Readback CRC needed to run.

7.1.2 Device Configuration Interface (DevC)

There is actually another interface layer between the PCAP and the processors which is called the Device Configuration interface (DevC). The processors communicate to the DevC interface, and the DevC interface forms a bridge with the PCAP interface [43]. As described in Section 7.1.1, the PCAP side interfaces directly with the configuration module.

The DevC interface is simply a dedicated address space of accessible memory-mapped registers that allow software to perform configuration operations (see Figure 7.1 for datapaths). Configuration scrubbing command sequences are first processed through the DevC interface which are then converted into PCAP protocol and subsequently transferred through the PCAP to the PL [43]. Appendix B gives the sequence for the exact PCAP transfer process. The DevC interface registers facilitate scrubbing in a number of ways:

1. Each data transfer from the PS to the PL is initiated by simply writing four registers with the parameters of the transfer in a specific order.

[1] devcfg.CTRL[PCAP_PR] must be set to a '1' to use the PCAP.

2. A number of status and interrupt bits are dedicated for communicating configuration errors, transfer errors, or internal PL errors. The status registers are valuable in providing debugging information when configuration transfers fail.

3. Interrupts are also used for reporting non-error conditions such as the DONE status of PCAP and DMA transfers. This allows the PS to know when to resume its program and when to count on the requested PL data being available.

4. Data transfers require exact widths to be specified. If mismatches occur (i.e. requesting a certain number of readback frames, but a different number actually gets sent back), the user will know why the transfer failed.

As previously mentioned, the PS can host an OS. Depending on the OS used and if the caches are enabled, writes to memory mapped registers (including all of the DevC registers used to perform PCAP transfers) will sometimes be cached and will not actually be written through to memory. The caches of the ARM processors are write-back [43]. Hence, the caches must be flushed if they are enabled any time a memory-mapped register is written to. Otherwise, transfers may not actually be initiated at all.

The command sequences and the scrubbing "golden" data (discussed in Section 6.2.1) are stored in large memories like DDR. The DMA controller can easily transfer these command sequences through the DevC and PCAP interfaces into the PL. Reads from the configuration memory (either readbacks or configuration register reads) are stored in contiguous blocks or software variables in DDR or local CPU registers, respectively. The scrubber can read and write to the configuration registers and to the configuration memory itself through these DevC/PCAP transfers.

7.1.3 PCAP Limitations

There are some limitations when performing PCAP readback transfers that must be understood when performing readback scrubbing. When a readback is requested, the returning data comes at a constant rate from the PL whether or not the PCAP receiver FIFO is ready for it. To prevent overflow of the receiver FIFO, the PCAP must transfer this data from the receiver FIFO to the destination memory via DMA over the PS AXI

interconnect faster than the configuration module can fill up the receiver FIFO. The data rate is determined by a combination of the PCAP clock rate and the PS AXI interconnect.

When reading back continuously, the DMA controller could hang and freeze the AXI bus if too many frames are being read or if the frames are being read back too fast. Two solutions to handle this issue are presented here. The first is to read smaller amounts of data (i.e. fewer frames) with long delays (on the order of milliseconds) between each read transfer. The other solution is to slow the PCAP clock (the default PCAP CLK is 100 MHz). The PCAP CLK can be slowed by writing to a System Level Control Register[1] (SLCR) with a larger clock divisor value.

Another limitation of PCAP readback is that a single readback request cannot be split over multiple DMA accesses. Sending a command requesting 606 words of readback data cannot be followed by a read of 404 words, then another read of 202 words. It must read all 606 words in one transfer. The implications of this behavior is that care must be taken when specifying source and destination lengths of the data transfer (see Appendix B).

Finally, due to hardware restrictions, all DMA transactions must be organized such that they do not cross a 4 KB boundary. Since readbacks request the number of desired frames plus one dummy frame for the frame buffer per transaction (see Section 3.4.2), the most data that the PS can request from the PL in a single transfer are 9 configuration frames + 1 dummy frame = 10 frames. It will be less than 10 frames if the transfer is reading data that crosses a row boundary (see Section 6.1.2). Ten frames equates to 1010 words or 4040 bytes, which is just under the 4096 byte (4 KB) boundary. Attempting to read more than 10 frames in a single transfer will result in DMA transfer errors.

7.2 Zynq Readback Scrubber

The first scrubbing architecture implementation is a readback scrubber. This scrubber uses the readback architecture presented in Section 6.4. Recall that the readback scrubber described in that section continuously reads back data, compares the data to a golden file (usually stored in DDR memory), and writes over corrupted frames if any upsets are detected.

[1] PCAP_CLK_CTRL bits[13:8] control clock divisor. The default divisor is 5. [43]

The readback scrubber does not consider the upset type scenarios explained in Chapter 4 since the Readback CRC is not being used.

The readback and hybrid (see Section 7.3) scrubbers were implemented on a XZC07020 FPGA running the Arch Linux ARMv7 distribution. The scrubber application running in Linux depends on the Xilinx Device Configuration driver (xdevcfg) to interact with the PCAP.

The xdevcfg is a kernel module that can be embedded in the kernel or loaded in at runtime. As a kernel module, it can do many things that would otherwise be difficult or impossible for userspace applications (i.e. the scrubber). It registers interrupt handlers for the PCAP, allocates contiguous chunks of physical memory for DMA, configures the FPGA clocks, and handles read and write transactions over DMA between the PCAP and userspace applications. These features make the module a clean interface to the PCAP that any applications can use without worrying about hardware details. Configuring the bitstream is possible using the xdevcfg module as provided by Xilinx, but the module must be modified in order to allow periodic configuration operations to occur without resetting the FPGA as well as to fix a few bugs in the module itself. If no OS is used, the PCAP can be used as well in bare metal to perform these same functions.

The performance metrics of the scrubbing architectures presented in this chapter were measured on XZC07020 FPGAs. It takes the Readback Scrubber approximately 1.82 to 10 seconds (T_D) depending on the processor load to scan through the entire device reading back 9 data frames at a time. These measurements equate to a T_d between 0.229 ms and 1.26 ms depending on the processor load. The processor load refers to the amount of application traffic on the PS AXI interconnect which can slow down the overall memory bandwidth (particularly for the DMA controller). The latency of scrubbing one frame, T_c, is measured to be between 0.1 ms and 1.86 ms depending on the processor load. The total scrubbing time for the readback scrubber, assuming $u = 1$ frame with an upset, is

$$\begin{aligned} T_s &= T_d * N + u * T_c \\ &= 2.29 \times 10^{-4} * 7942 + 1 * 1.0 \times 10^{-4} \\ &= 1.82 \text{ seconds} \end{aligned} \tag{7.1}$$

for a low processor load. For a high processor load the total scrubbing time is

$$\begin{aligned} T_s &= T_d * N + u * T_c \\ &= 1.26 \times 10^{-3} * 7942 + 1 * 1.86 \times 10^{-3} \\ &= 10.009 \text{ seconds.} \end{aligned} \tag{7.2}$$

This architecture has already been integrated with the CHREC Space Processor (CSP), a work which combines an FPGA and processing system approach to embedded space computing. The CSP is intended to be an affordable yet reliable data processing option. CSP boards will be used in future NASA Goddard satellite missions (see Figure 7.2). One of these missions is the STP-H5/ISEM and another is the CeREs mission [4].

Figure 7.2: CHREC Space Processor [4]

7.2.1 Performing Readback on the Zynq

Each readback performed by the scrubber on the Zynq requires two PCAP transfers: the first is to request the readback data, and the second is to actually receive the data from the PCAP. While the PCAP has sufficient bandwidth capabilities to write bitstreams to the configuration without problems, the readback hardware is smaller (i.e. the receiver FIFO is

smaller than the transmit FIFO; see Figure 7.1) and thus more limited. Timing considerations must be adhered to when implementing a readback scrubber on a Zynq device [4].

7.2.2 Readback Radiation Testing

Because it is expensive to put FPGAs directly into space, scrubbers are first validated in ionizing-radiation beams at specialized facilities on Earth. These facilities contain particle accelerators that can create focused beams of heavy-ions, protons, neutrons and others particles used for high-radiation experiments. It is useful to validate scrubbing architectures by placing FPGAs in front of radiation beams and observing their tolerance of direct radiation effects and their ability to detect and correct upsets. Such experiments were performed with the readback scrubbing architecture running on a Zynq XZC07020[1] board on two different occasions (September 2014 and May 2015) both at the TRIUMF facility in Vancouver, Canada. The TRIUMF facility has multiple irradiation facilities, but the radiation test described here was performed in the BL1B beam. The spectrum for this beam is a 1/E falloff of flux with an energy up to 500 MeV [61]. The fluence for the September 2014 test was $3.47 * 10^9$ neutrons/cm^2/s. The fluence for the May 2015 test was $1.49 * 10^7$ neutrons/cm^2/s.

The radiation test setup consisted of positioning the FPGA in front of the beam near the beam's aperture (see Figure 7.3; the other side of the red square is where the Zynq SoC running the scrubber is located). Ethernet and micro-usb cables were used to monitor the FPGA via a UART into a terminal connected to a laptop in a protected room. The XZC07020 SoC ran Linux, which included an SSH server to allow easy access to the scrubbing software program. Software watchdogs, which monitor whether the processor stalls for a long period of time, were also implemented to automatically reboot the OS in the event that the processor cores themselves incurred fatal radiation upsets.

The results from these tests consist of extensive log output which verify the correct operation of the readback scrubber. Key data observed includes unique upset types that the scrubber detected and corrected as well as how many of each type were found. The number

[1]Some versions of silicon are not guaranteed to function properly for scrubbing purposes [60]. For the XZC07020 board, the required silicon version must be revision D production silicon version 2 or higher.

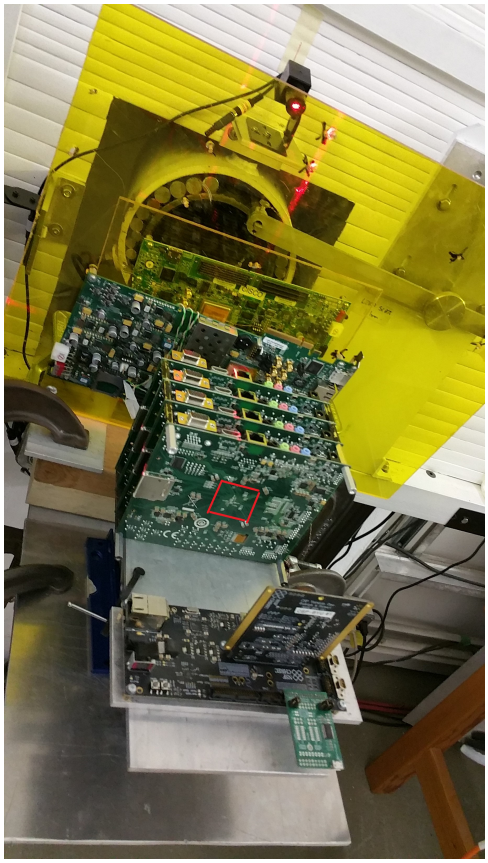

Figure 7.3: Radiation Beam Test Setup at TRIUMF

of total readback scans and reboots was also recorded. Table 7.1 shows the results from the test performed in September 2014. Table 7.2 shows the results from the test performed in May 2015. Note that the amount of beam time that each of these FPGAs received is not certain, though the September 2014 test did have more beam time than did the May 2015 test (evidenced by the number of Readbacks performed and upsets corrected).

The data logged by the scrubbers contain interesting observations about the behavior of radiation upsets. For instance, it is interesting to note that almost all of the MBUs occurred in adjacent frames [48]. In the September 2014 data, one of the four-bit and five-bit upsets occurred in the same word and bits but in different frames. These adjacent MBUs were almost certainly caused by the same SEU that affected the interleaving bits of separate frames (see Section 4.1.3). Appendix G gives a sample full readback scrub log excerpt.

Table 7.1: TRIUMF September 2014 Readback Scrubber Radiation Test Results

Number of Upsets in a Frame	Occurrences
1	874
2	63
3	8
4	2
5	3
Total Upsets Corrected	1047
Readbacks	99539
Reboots	157

Table 7.2: TRIUMF May 2015 Readback Scrubber Radiation Test Results

Number of Upsets in a Frame	Occurrences
1	375
2	21
3	2
5	1
6	1
Total Upsets Corrected	434
Readbacks	352
Reboots	80

7.3 Zynq Hybrid Scrubber

The culminating work of this thesis is presented in this section with the implementation of the Hybrid Scrubbing Architecture introduced in Section 6.5. This hybrid scrubber is also implemented on a Zynq SoC which utilizes the 7-Series Readback CRC hardware to correct SBUs and detect all upsets. The hybrid scrubber performs readback scrubbing on a single frame if MBUs are detected in that frame. If a CRC-Only error arises, then a full readback scrub is performed across the entire device as though it were a normal readback scrubber.

7.3.1 Hybrid Hardware

By allowing the Readback CRC to handle upset detection, a significant bandwidth load can be lifted from the processor. Where the readback scrubber occupies one core at

around a 57 percent utilization, the polling of the hybrid scrubber uses a mere 5 percent of one core[1]. The delay between polling or readbacks affects the average processor utilization. Performing readbacks through the PCAP, however, is significantly more bandwidth-taxing than polling for the Readback CRC output.

Recall from Section 6.5.1 that the the hybrid scrubbing architecture requires a mechanism for both reading the values of the `FRAME_ECCE2` primitive and loading those values into a software-readable register to be read by the processor. The Zynq devices allow for custom AXI-bus peripherals to easily be created and connected to memory-mapped registers that are readable in software. One such peripheral was created to instantiate the `FRAME_ECCE2` primitive and additional logic to send detected upsets to the software scrubbing program. As shown in Figure 7.4, there are many small and simple components used to accurately deliver the batches of `FRAME_ECCE2` signals to the memory-mapped registers of the AXI-bus peripheral. All of the hardware presented below outside of the `FRAME_ECCE2` primitive must operate at a clock frequency greater than or equal to that of the `FRAME_ECCE2`.

1. **Frame ECC Latch and Upset Detection Logic**

   The Frame ECC latch holds the values of the `FRAME_ECCE2` signals whenever SYNDROMEVALID pulses. The upset detection logic detects if SYNDROME holds a nonzero value (indicating that the ECC detected an error). These signals are then saved together as a batch and loaded into a FIFO (see Chapter 6).

2. **FIFO**

   The software program obviously runs at a much slower rate than the Readback CRC hardware. This fact results in the necessity of buffering the batches of `FRAME_ECCE2` signals between the hardware and software. A small FIFO is used (usually about 32 entries deep), to store the `FRAME_ECCE2` batches to be subsequently read in software via the memory-mapped registers.

3. **Memory-Mapped Registers**

   The output of the FIFOs feeds directly into a set of memory-mapped registers. Because

[1]These measurements were taken using the **top** Linux program running both the readback and hybrid scrubbers for 30 minutes and observing the average load over 15 minutes metric. These values do fluctuate over time.

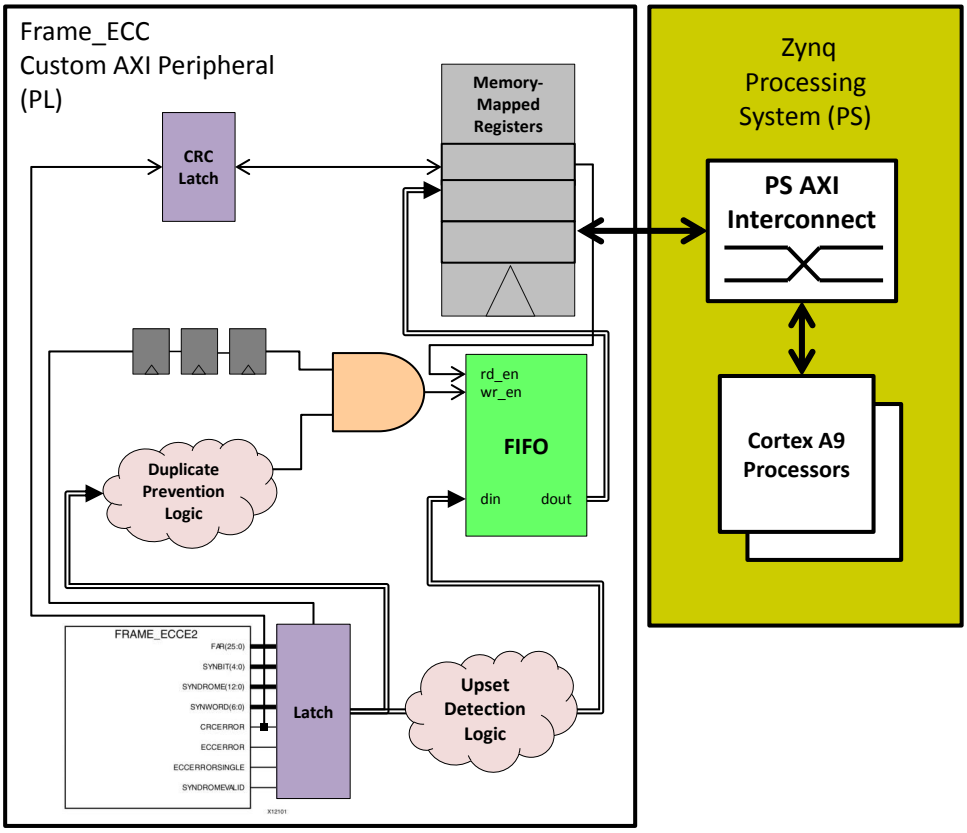

Figure 7.4: Hybrid Hardware for Zynq

they are memory-mapped, they have a dedicated address space in software. A scrubbing program can simply dereference an address (specified by the mapping), to obtain the output of the FIFO. A handshaking protocol must be followed to accurately pop entries out of the FIFO once they have successfully been read by the software. Typically this is done via another memory-mapped register which is write-only from the software and whose bits are wired into the FIFO's read-enable signal.

Another option besides an AXI-bus peripheral is using a JTAG BSCANE primitive. The BSCANE can imitate the behavior of memory-mapped registers to load the FIFO's output, followed by software reading that output using JTAG commands. For hybrid scrubber implementations on FPGAs other than the Zynq SoC that use the JTAG interface to access the configuration, the BSCANE is the ideal solution to fulfill this role.

4. **CRC Latch**

   Recall from Section 4.2.1 that the CRCERROR is asynchronous to the other `FRAME_ECCE2` signals. It requires a direct route to the memory-mapped registers (bypassing the FIFO), so that it can be read in software as quickly as possible. Similar to asserting the FIFO's read-enable signal, to acknowledge the CRCERROR signal, another bit in the software write-only register is designated to be a latch reset signal. The latch reset bit is written by the software to clear the CRC latch, allowing new CRC errors to be detected.

5. **Duplicate Prevention Logic**

   Recall from Chapter 4 that the Readback CRC will generate an infinite number of `FRAME_ECCE2` batches under the CORRECT_AND_CONTINUE mode for some upset types. To prevent the FIFO from filling up with duplicate upset data (since the Readback CRC continuously scans faster than the software reads from the FIFO), a duplication prevention circuit is used. The duplicate prevention circuit must correctly identify when duplicate entries are occurring, while simultaneously allowing entries that may differ only slightly (i.e. same EFAR but different SYNDROMEs).

Table 7.3: FPGA Resource Utilization Comparison of Hybrid Scrubber and Xilinx SEM IP

XZC07020	LUTs	%	FFs	%	BRAM (18 KB)	BRAM (36 KB)	%
SEM IP	838	1.6	682	0.64	3	3	3.2
Hybrid	357	0.67	740	0.70	0	1	0.7

Table 7.3 shows a comparison of the resource utilization of the hybrid hardware in the Zynq XZC07020 FPGA compared to the amount used by the Xilinx SEM IP [3]. Note that the Xilinx SEM IP is already considered to use a small hardware footprint. Thus, the hardware footprint of this hybrid scrubber is also quite minimal.

The hardware required for the hybrid scrubber would logically be three times larger if it were triplicated into a TMR circuit. The one single-point-of-failure that cannot be triplicated is the connection to the `FRAME_ECCE2` primitive because only one Frame ECC

primitive exists per 7-Series FPGA device. Further research will need to be performed to examine the failure of the hybrid hardware, as no conclusive evidence has been observed to indicate that the hardware incurred unrecoverable upsets in the radiation tests.

7.3.2 Hybrid Scrubber Performance

Due to the high performance of the Readback CRC in upset detection and SBU correction, the hybrid scrubber's performance can be measured in terms of upset detection, SBU correction, and MBU correction. These metrics were measured both in software by using timestamps between functions and in hardware by counting clock cycles between error events.

For SBUs, the T_D for the Readback CRC is 8.02 ms, which means that

$$\begin{aligned} T_d &= T_D/N \\ &= 8.02/7942 \\ &= 1\ \mu s \end{aligned} \tag{7.3}$$

per frame. T_c, the time to correct an SBU for the Readback CRC, is measured to be 4.5 μs.

The type of MBU affects the T_D for the Readback CRC. Assuming that the MBU is the only upset, the Readback CRC detects two-bit and larger even-numbered upsets detected by the ECC within 8.02 ms, the T_D stated previously. Odd-numbered out-of-bounds are also detected within 8.02 ms (because the SYNWORD immediately indicates an invalid word). Odd-numbered in-bounds MBUs are detected within 16.04 ms due to the incorrect repair that the Readback CRC inserts in its attempt at satisfying the ECC (see Section 4.1.2). Odd MBUs with masked-bit correction upsets take about 24.06 ms (3 passes of the Readback CRC) to detect.

The correction speed, T_c, for MBU correction is the same as the that of readback scrubber of Section 7.2 which is between 0.1 ms and 1.86 ms depending on the processor load. Because there is some overhead between the times that the Readback CRC detects the MBU and the time that the hybrid scrubber begins to scrub the frame, another parameter

T_o must be added to the total scrubbing time equation:

$$T_s = N * T_d + u * T_c + T_o. \tag{7.4}$$

Table 7.4 shows the performance metrics for the hybrid scrubber assuming that the upset type given in the first column is the only upset that has occurred ($u = 1$) across the entire device.

Table 7.4: Performance Metrics for Hybrid Scrubber

Upset Type	T_D **(ms)**	T_c **(ms)**	T_o **(ms)**	T_s **(ms)**
SBUs	8.02	0.0045	0	8.024
Two-Bit Upsets	8.02	1.86	3.5	13.38
Odd-Numbered Out-Of-Bounds MBUs	8.02	1.86	3.5	13.38
Odd-Numbered In-Bounds MBUs	16.04	1.86	3.5	21.40
Masked Bit Upsets	24.06	1.86	3.5	29.42
Even-Numbered MBUs	8.02	1.86	3.5	13.38
CRC-Only (Readback Scrubber)	1,820.0	1.86	-	1,821.86

In the case of a CRC-Only error, all of the `FRAME_ECCE2` FIFO entries have been popped off, yet an error still exists. It is not really possible or useful to measure consistent values for T_o in this situation. The correction solution is to perform a full readback scrub across the entire device. The latency of that operation as given in Section 7.2 is between 1.82 and 10.009 seconds depending on the processor load. Again, the processor load refers to the amount of application bus traffic on the PS AXI interconnect which can slow DMA transfers. Because every frame must be checked when a CRC-Only error occurs, $1.822 \leq T_s \leq 10.009$ seconds. The CRC-Only error correction method is equivalent to the readback scrubber of Section 7.2, so Table 7.4 can also be used to compare the performance of the hybrid and readback scrubbers.

7.3.3 Hybrid Radiation Testing

In October 2015, the hybrid scrubber was tested in a neutron radiation beam at Los Alamos Neutron Science Center (LANSCE) [62]. The test was performed in the Weapons

Neutron Research (WNR) facility in a beam which has a neutron energy spectrum from 0.1 MeV to 600 MeV [63] using the Target Flight Path 30L (ICE House). The fluence of this test was $1.01 * 10^{11}$ neutrons/cm^2/s.

Like the readback test setup, the Zynq board hosting the hybrid scrubber was placed near the beam aperture in between other FPGAs being used in the radiation test. The hybrid scrubber received more beam time (152,168.03 seconds $\approx$ 42.26 hours) than what the readback scrubber tests in Section 7.2.2 received. Table 7.5 gives the results of the hybrid scrubber's execution in detecting and correcting upsets.

Table 7.5: LANSCE 2015 Hybrid Scrubber Radiation Test Results

Number of Upsets in a Frame	**Occurrences**
1	4,239
2	326
3	133
4	41
5	3
6	7
7	2
8	1
14	1
16	1
Total Upsets Corrected	5,563
Readbacks	69
Reboots	281
Even $\rightarrow$ Odd Upset Scenario	71
Odd out-of-bounds Upsets	0
Masked Bit Upsets	0
Total Runtime	152,168.03 seconds

The hybrid scrubber was successful in detecting and correcting many kinds of upsets. Similar to the Readback Scrubber's data, the majority of the MBUs were observed in adjacent frames. The frame interleaving (see Section 4.1.3) helped to break up some MBUs into SBUs, but oftentimes, one large MBU would be broken up into two smaller MBUs in different

frames. The 16-bit upset, for instance, occurred in a frame two FRADs sequentially after the 14-bit upset in the same word and involving most of the same bits.

Almost all of the odd-numbered upsets became even-numbered upsets due to the incorrect repair (see Section 4.3.4). None of the odd-numbered MBUs resulted in out-of-bounds or masked bit upsets (see Section 4.3.2). This observation is most likely due to the fact that most MBUs often occurred near each other in word and bit locations. To generate an out-of-bounds SYNWORD, the upset bits have to be in randomly distributed locations, not clumped up in the same word or pair of words. Masked bit upsets are rarely seen in fault injection; thus, it is not surprising to not have witnessed any in the radiation beam. Interestingly, the Even-Numbered MBU $\rightarrow$ Odd-Numbered MBU/SBU upset scenario occurred 71 times, with almost every instance being the result of an MBU overlapping two interleaved frames. The scrubber anticipated this behavior, however, and was able to report it accordingly.

It is also important to note that for both the hybrid and readback scrubbers, the processor failed on a number of occasions (indicated by the number of reboots). Again, processor reliability techniques are essential to maintain the overall reliability of a Zynq system. The susceptibility of the kernel to radiation effects cannot be ignored.

In terms of efficiency, however, the hybrid scrubber performed a full readback scrub only 69 times. Compared to the Readback Scrubber results of Table 7.1, which performed a full readback scrub 99,539 times, this observation denotes a notable improvement in efficiency. The hybrid scrubber did run for a longer time, but it corrected more upsets in fewer readbacks via the PCAP.

The Zynq SoC is a landmark FPGA featuring dual-core ARM processors that allow a scrubbing program to be implemented primarily in software. Two implementations of 7-Series Scrubbing Architectures on Zynq SoCs were presented in this chapter. These scrubbers were validated in radiation beam tests.

Chapter 8

Conclusion

FPGAs are playing an increasingly important role in performing computationally-intensive processing applications in high-radiation environments. Due to the susceptibility of an FPGA's configuration memory to radiation particle strikes, upset mitigation techniques must be employed to ensure the reliability of the user design. This work explored the contribution of configuration scrubbing as an essential upset mitigation technique to help improve reliability of FPGA systems, particularly when used with TMR.

While several scrubbing architectures already exist for previous Xilinx FPGA families, very few architectures exist for 7-Series devices. This thesis set out to introduce configuration scrubbing architectures for Xilinx 7-Series SRAM-based FPGAs that achieve the desirable combination of high-performance in detecting and correcting upsets, small hardware footprints, and a high level of robustness demonstrated by handling large and complex upset types. The novel hybrid scrubbing architecture presented in this thesis represents the realization of these objectives.

The hybrid and readback scrubbing architectures introduced in this thesis were implemented on a Zynq SoC and were validated in beam tests at LANSCE and TRIUMF radiation testing facilities. The hybrid scrubber detected and corrected MBUs with as many as 16 upset bits in a single frame, while the readback scrubber detected and corrected MBUs with as many as six upset bits in a single frame.

The hybrid scrubber on the Zynq SoC utilizes the Readback CRC to achieve fast detection performance. SBUs and even-numbered MBUs are detected in 8.02 ms. Odd-numbered out-of-bounds MBUs are also detected in 8.02 ms. Odd-numbered in-bounds MBUs are detected in 16.04 ms. The hybrid scrubber also achieves fast correction speeds. It takes approximately 1.86 ms to readback scrub a single frame which corrects all even and

odd-numbered upsets in that frame. The Readback CRC handles SBUs correcting them in 4.5 μs, for a total scrubbing time of 8.024 ms. When a CRC-Only error occurs requiring a full readback scrub of the entire device, the scrubber accomplishes this task in 1.82 seconds with a low processor load. The total scrubbing time of even-numbered and odd-numbered out-of-bounds MBUs is 13.38 ms, and is 21.40 ms for odd-numbered in-bounds MBUs. The readback scrubber corrects all upset types in 1.82 seconds with a low processor load.

The detection and correction performance of these scrubbers is crucial in helping to improve the reliability of the system. When TMR is applied to a user design, having a much faster repair rate versus upset rate results in significant reliability and MTTF improvements. When taking into account the expected upset rates of high-radiation environments like space, the performance metrics of these scrubbers indicate that a substantial improvement in reliability for an FPGA user design would be observed compared to not using TMR and scrubbing.

The innovative Zynq SoC family presented a new opportunity for developing scrubbing architectures. The dual-core Cortex-A9 ARM processing system supports a software implementation of the scrubbing logic which frees up precious FPGA resources otherwise needed to implement an internal scrubbing solution. The PCAP facilitates software and hardware interaction by allowing smooth configuration memory transactions. Such advantages offer scrubbers an easier debug environment, improved portability, and shorter compilation times.

The readback and hybrid scrubbing architectures presented in this thesis are scheduled to go up into space on multiple NASA satellite missions as part of the CHREC Space Processor (CSP). The CSP is a low-cost, high-speed solution for space processing applications built around a Zynq XZC07020 FPGA [64]. One of the missions, STP-H5/ISEM, is a technology mission on the International Space Station where two CSPs will be paired together to evaluate hardware performance in a space setting [4]. Another NASA mission, the CeREs, is a heliophysics science mission in which a single CSP will be performing data processing. The scrubbers on these and future missions are responsible for maintaining the fault tolerance of the FPGA configuration memory.

Countless other opportunities exist for these scrubbers as FPGAs in high-radiation environments like space become more commonplace. High-energy physics experiments that require high-speed intensive data processing are also ideal candidates for FPGAs running these scrubbers. For example, an adapted version of the readback scrubbing architecture of this thesis was used to mitigate upsets for a Multi-Gigabit Transceiver (MGT) experiment in a proton beam that tested the radiation tolerance in high-speed data transmission [65]. The scrubbers developed in this thesis can be adapted to help mitigate upsets on FPGAs in future radiation test experiments as well.

Future Work

This work presented scrubbing architectures for the 7-Series devices. While these scrubbers operate effectively and efficiently, there is still room for improvements. For example, making the hybrid scrubber interrupt-driven instead of poll-driven could prove to be a beneficial enhancement. Furthermore, as these scrubbers undergo more and more radiation beam tests and random constrained fault injection experiments, new types of error scenarios may be discovered. The scrubbing architectures will have to be adapted to handle those situations successfully.

Just as this thesis presented scrubbing architectures for a newer FPGA device family, the Xilinx 7-Series, each new generation of FPGAs will require corresponding scrubbing architectures to be developed that detect and correct configuration upsets for those FPGAs. The next step for FPGA configuration scrubbing will be migrating these present 7-Series techniques to the recently-released Xilinx UltraScale FPGA boards.

The UltraScale and UltraScale+ FPGA families are fabricated with 20nm and 16nm processing technology respectively [66]. The Zynq UltraScale+ features a 64-bit quad-core ARM Cortex-A53 processor that achieves 5X system performance improvement over the Zynq device presented in this thesis. Scrubbing on the UltraScale will be somewhat different than 7-Series scrubbing. Frames are now 123 words instead of 101 words, and three of those words appear to be separate ECC values (compared to only one ECC value in a 7-Series frame). Most likely the combination of these ECCs will offer more than a SECDED code. Furthermore, the `FRAME_ECCE3` is "reserved" by Xilinx [30], making the details of each of its

signals harder to understand and use properly when creating a hybrid scrubber. However, having the knowledge presented in this thesis provides a useful foundation for anticipating the behaviors and operations of these components despite the differences with their 7-Series equivalents.

This work has demonstrated the usefulness of configuration scrubbing in detecting and correcting configuration memory upsets in high-radiation environments. As the process technology that is used to fabricate FPGAs continues to shrink, the need for configuration scrubbing will increase. When used with TMR, configuration scrubbing will continue to offer significant reliability improvements and become an integral feature of the fault tolerance mechanisms of many computationally intensive processing systems used in high-radiation environments.

Acronyms

ARM Advanced RISC Machines.

AXI Advanced eXtensible Interface protocol.

BRAM Block Random Access Memory.

BSCANE Boundary Scan Primitive.

CHREC NSF Center of High-Performance Reconfigurable Computing.

CLB Configuration Logic Block.

CMD Command Register.

COR1 Configuration Options Register 1.

COTS Commercial-off-the-Shelf.

CPU Central Processing Unit.

CRAM Configuration Random Access Memory.

CRC Cyclic Redundancy Check.

CSP CHREC Space Processor.

CTL0 Control Register 0.

DDR Double Data Rate Dynamic RAM.

DevC Device Configuration Interface.

DMA Direct Memory Access.

DMR Dual Modular Redundancy.

DRP Dynamic Reconfiguration Port.

DSP Digital Signal Processing.

ECC Error Correction Code.

EFAR Error Frame Address.

FAR Frame Address Register.

FDRI Frame Data Register In.

FDRO Frame Data Register Out.

FF Flip-Flop.

FIFO First in, First Out.

FIT Failure in Time.

FPGA Field Programmable Gate Array.

FRAD Frame Address.

FSM Finite State Machine.

FWFT First Word Fall Through.

GLUTMASK Global LUT Mask.

HDL Hardware Description Language.

HWICAP Hardware ICAP IP.

ICAP Internal Configuration Access Port.

iMPACT Xilinx software tool that uses JTAG to perform configuration operations.

JTAG Joint-Test Action Group.

LANSCE Los Alamos Neutron Science Center.

LUT Look-up-Table.

Mb Megabit.

MBU Multi-Bit Upset.

MCU Multi-Cell Upset.

MGT Multi-Gigabit Transceiver.

MTTD Mean Time to Detection.

MTTF Mean Time to Failure.

MTTR Mean Time to Repair.

OS Operating System.

PCAP Processor Configuration Access Port.

PIP Programmable Interconnection Points.

PL Programmable Logic.

PS Processing System.

RCFG Read Configuration Command.

RHBD Radiation Hardened By Design.

RHBP Radiation Hardened By Process.

SBU Single-Bit Upset.

SDK Software Development Kit.

SECDED Single Error Correction, Double Error Detection.

SEM IP Soft-Error Mitigation Intellectual Property.

SEU Single Event Upset.

SLCR System Level Control Register.

SoC System-on-Chip.

SRAM Static Random Access Memory.

SRL Shift Register LUT.

SSH Secure Shell.

$\mathbf{T_c}$ Time to correct an upset in a single frame.

$\mathbf{T_d}$ Time to detect an upset in a single frame.

$\mathbf{T_D}$ Upper bound time to detect upsets in exactly one frame.

$\mathbf{T_s}$ Total scrubbing time.

TMR Triple Modular Redundancy.

TMRR TMR with Repair.

TRIUMF Canada's National Laboratory for Particle and Nuclear Physics.

UCF User Constraints File.

WCFG Write Configuration Command.

XADC Analog to Digital Converter.

xdevcfg Xilinx Device Configuration kernel module driver.

Bibliography

[1] Xilinx, *7 Series FPGAs Configuration User Guide*, 2015, UG470 (v.1.10). [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf ix, xi, 9, 17, 19, 21, 23, 25, 26, 29, 30, 31, 32, 33, 45, 46, 48, 50, 77, 94, 129, 134

[2] Xilinx, *Vivado Design Suite 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide*, 2015, UG953 (v.2015.2). [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug953-vivado-7series-libraries.pdf xi, 23, 41

[3] Xilinx, *Soft Error Mitigation Controller*, 2014, PG036 (v4.1). [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/sem/v4_1/pg036_sem.pdf xii, 2, 12, 24, 27, 32, 36, 40, 46, 55, 69, 70, 71, 76, 78, 83, 104

[4] D. Rudolph, C. Wilson, J. Stewart, P. Gauvin, A. George, H. Lam, G. Crum, M. Wirthlin, A. Wilson, and A. Stoddard, "CSP: A Multifaceted Hybrid Architecture for Space Computing," 2014. xii, 1, 92, 98, 99, 110

[5] R. Santos, S. Venkataraman, and A. Kumar, "Dynamically adaptive scrubbing mechanism for improved reliability in reconfigurable embedded systems," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, June 2015, pp. 1–6. 1, 67

[6] H. Michel, A. Belger, T. Lange, B. Fiethe, and H. Michalik, "Read back scrubbing for SRAM FPGAs in a data processing unit for space instruments," in *Adaptive Hardware and Systems (AHS), 2015 NASA/ESA Conference on*, June 2015, pp. 1–8. 1, 23

[7] F. Bubenhagen, B. Fiethe, T. Lange, H. Michalik, and H. Michel, "Reconfigurable platforms for data processing on scientific space instruments," in *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, June 2013, pp. 63–70. 1

[8] A. Ebrahim, T. Arslan, and X. Iturbe, "On enhancing the reliability of internal configuration controllers in FPGAs," in *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, July 2014, pp. 83–88. 1, 66

[9] F. Brosser, E. Milh, V. Geijer, and P. Larsson-Edefors, "Assessing scrubbing techniques for Xilinx SRAM-based FPGAs in space applications," in *Field-Programmable Technology (FPT), 2014 International Conference on*, Dec 2014, pp. 296–299. 1, 2, 3, 15, 69

[10] M. Wirthlin, "High-Reliability FPGA-Based Systems: Space, High-Energy Physics, and Beyond," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 379–389, March 2015. 1, 5, 6, 8

[11] S. Lee, R. Kjar, J. Peel, and G. Kinoshita, "Radiation-Hardened Silicon-Gate CMOS/-SOS," *Nuclear Science, IEEE Transactions on*, vol. 24, no. 6, pp. 2205–2208, Dec 1977. 2, 11

[12] Xilinx, *Radiation Hardened, Space-Grade Virtex 5QV Family Overview*, 2014, DS192 (v.1.4). 2, 11, 66

[13] M. Berg, C. Poivey, D. Petrick, D. Espinosa, A. Lesea, K. LaBel, M. Friendlich, H. Kim, and A. Phan, "Effectiveness of Internal Versus External SEU Scrubbing Mitigation Strategies in a Xilinx FPGA: Design, Test, and Analysis," *Nuclear Science, IEEE Transactions on*, vol. 55, no. 4, pp. 2259–2266, Aug 2008. 2, 64

[14] (2015) 7 Series FPGAs Overview. Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf 2

[15] J. Heiner, N. Collins, and M. Wirthlin, "Fault Tolerant ICAP Controller for High-Reliable Internal Scrubbing," in *Aerospace Conference, 2008 IEEE*, March 2008, pp. 1–10. 2, 15, 65, 133

[16] A. Harding and M. Wirthlin, "Improving the Reliability of Xilinx 7 Series FPGAs through Configuration Scrubbing," in *20th Annual Fellowship Symposium Utah NASA Space Grant Consortium, Hill Aerospace Museum*, April 2014. 2, 40, 55, 69, 70

[17] I. Herrera-Alzu and M. Lopez-Vallejo, "Design Techniques for Xilinx Virtex FPGA Configuration Memory Scrubbers," *Nuclear Science, IEEE Transactions on*, vol. 60, no. 1, pp. 376–385, Feb 2013. 3, 16, 59

[18] H. M. Doss. (2015) Ionizing Radiation and Humans – The Basics. [Online]. Available: http://www.physicscentral.com/explore/action/radiationandhumans.cfm 5

[19] R. Baumann, "Soft errors in advanced semiconductor devices-part I: the three radiation sources," *Device and Materials Reliability, IEEE Transactions on*, vol. 1, no. 1, pp. 17–22, Mar 2001. 6

[20] F. Wang and V. Agrawai, "Single Event Upset: An Embedded Tutorial," in *21st International Conference on VLSI design*, 2008. 6

[21] R. Katz. (2010) A scientific study of the problems of digital engineering for space flight systems, with a view to their practical solution. [Online]. Available: http://klabs.org/DEI/References/design_guidelines/content/guides/nasa_asic_guide/Glossary.html 6, 9

[22] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba, "Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule," *Electron Devices, IEEE Transactions on*, vol. 57, no. 7, pp. 1527–1538, July 2010. 7

[23] X. Iturbe, M. Azkarate, I. Martinez, J. Perez, and A. Astarloa, "A novel SEU, MBU and SHE handling strategy for Xilinx Virtex-4 FPGAs," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, Aug 2009, pp. 569–573. 7, 65

[24] N. Harward, M. Gardiner, L. Hsiao, and M. Wirthlin, "Estimating soft processor soft error sensitivity through fault injection," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, May 2015, pp. 143–150. 9, 32

[25] A. Sari and M. Psarakis, "Scrubbing-based SEU mitigation approach for Systems-on-Programmable-Chips," in *Field-Programmable Technology (FPT), 2011 International Conference on*, Dec 2011, pp. 1–8. 9, 67

[26] U. Legat, A. Biasizzo, and F. Novak, "Soft Error Recovery Technique for Multiprocessor SOPC," in *Test Symposium (ATS), 2011 20th Asian*, Nov 2011, pp. 175–180. 9, 66

[27] Xilinx, *Virtex-4 FPGA Configuration User Guide*, 2009, UG071 (v.1.11). [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug071.pdf 9, 35, 46

[28] Xilinx, *Virtex-5 FPGA Configuration User Guide*, 2012, UG191 (v.3.11). [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug191.pdf 9, 46

[29] Xilinx, *Virtex-6 FPGA Configuration User Guide*, 2014, UG360 (v.3.8). [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug360.pdf 9

[30] Xilinx, *UltraScale Architecture Configuration User Guide*, 2015, UG570 (v.1.3). [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug570-ultrascale-configuration.pdf 9, 111

[31] Xilinx, *Device Reliability Report*, 2015, UG116 (v.10.3.1). [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug116.pdf 9

[32] M. Gardiner, "An Evaluation of Soft Processors as a Reliable Computing Platform," Master's thesis, Brigham Young University, Provo, UT 84602, 2015. 11

[33] (2013) 7 Series FRAME_ECCE2 Port Descriptions and Functionality. Xilinx. [Online]. Available: http://www.xilinx.com/support/answers/54350.html 12, 42

[34] (2008) Regarding SEU detetction and correction. Xilinx. [Online]. Available: http://forums.xilinx.com/t5/Virtex-Family-FPGAs/Regarding-SEU-detetction-and-correction/m-p/161546/highlight/true#M10869 13

[35] M. Shooman, *Reliability of Computer Systems and Networks*, 2nd ed. Wiley, 2002. 14, 123

[36] D. McMurtrey, K. Morgan, B. Pratt, and M. Wirthlin, "Estimating TMR Reliability on FPGAs Using Markov Models," 2008, Paper 149. 14, 15

[37] N. Rollins, M. Fuller, and M. Wirthlin, "A Comparison of fault-tolerant memories in SRAM-based FPGAs," in *Aerospace Conference, 2010 IEEE*, March 2010, pp. 1–12. 14, 15

[38] G. S. G. Miller, C. Carmichael, *Single-Event Upset Mitigation for Xilinx FPGA Block Memories*, 2008, XAPP962 (v.1.1). 15

[39] A. Saleh, J. Serrano, and J. Patel, "Reliability of scrubbing recovery-techniques for memory systems," *Reliability, IEEE Transactions on*, vol. 39, no. 1, pp. 114–122, Apr 1990. 15

[40] Xilinx, *7 Series FPGAs Configurable Logic Block User Guide*, 2014, UG474 (v.1.7). [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf 17

[41] Xilinx, *Kintex-7 FPGAs Data Sheet: DC and AC Switching Characteristics*, 2015, DS182 (v.2.13). 23

[42] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb, "FPGA partial reconfiguration via configuration scrubbing," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, Aug 2009, pp. 99–104. 23, 34, 61, 62, 65, 66

[43] Xilinx, *Zynq-7000 AP SoC Technical Reference Manual*, 2015, UG585 (v.1.10). [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf 24, 91, 92, 94, 95, 96

[44] R. Le, *Soft Error Mitigation Using Prioritized Essential Bits*, 2012, XAPP538 (v.1.0). 32, 78

[45] Xilinx. Partial Reconfiguration in the ISE Design Suite. [Online]. Available: http://www.xilinx.com/tools/partial-reconfiguration.htm 34

[46] (2015) How can we infer a word in a frame from syndrome value (FRAME_ECCE2). Xilinx. [Online]. Available: http://forums.xilinx.com/t5/7-Series-FPGAs/How-can-we-infer-a-word-in-a-frame-from-syndrome-value-FRAME/td-p/558953 38

[47] Jameel Hussein and Gary Swift, *Mitigating Single-Event Upsets*, May 2015, WP395 (v.1.1). 40, 46, 84

[48] M. Wirthlin, D. Lee, G. Swift, and H. Quinn, "A method and case study on identifying physically adjacent multiple-cell upsets using 28-nm, interleaved and secded-protected arrays," *Nuclear Science, IEEE Transactions on*, vol. 61, no. 6, pp. 3080–3087, Dec 2014. 41, 100, 141

[49] A. Lesea. (2013) HCLK config bits/ECC bits. [Online]. Available: http://forums.xilinx.com/t5/Implementation/HCLK-config-bits-ECC-bits/td-p/337833 44, 84

[50] (2010) SP601 Post Configuration CRC. Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/xtp082.pdf 44

[51] Xilinx, *Constraints Guide*, 2013, UG625 (v.14.5). [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/cgd.pdf 47

[52] U. Legat, A. Biasizzo, and F. Novak, "SEU Recovery Mechanism for SRAM-Based FPGAs," *Nuclear Science, IEEE Transactions on*, vol. 59, no. 5, pp. 2562–2571, Oct 2012. 66

[53] G. Nazar, L. Santos, and L. Carro, "Accelerated FPGA repair through shifted scrubbing," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013, pp. 1–6. 67

[54] S. Venkataraman, R. Santos, S. Maheshwari, and A. Kumar, "Multi-directional error correction schemes for SRAM-based FPGAs," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014, pp. 1–8. 67

[55] S. Venkataraman, R. Santos, A. Das, and A. Kumar, "A bit-interleaved embedded hamming scheme to correct single-bit and multi-bit upsets for SRAM-based FPGAs," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014, pp. 1–4. 68

[56] M. Ebrahimi, P. Rao, R. Seyyedi, and M. Tahoori, "Low-Cost Multiple Bit Upset Correction in SRAM-Based FPGA Configuration Frames," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015. 68

[57] A. Harding and M. Wirthlin, "Hybrid Configuration Scrubbing for Xilinx Series-7 FPGAs," in *First International Workshop on FPGAs for Aerospace Applications (FASA 2014)*, August 2014. 70

[58] (2015) Soft Error Mitigation (SEM) Core. Xilinx. [Online]. Available: http://www.xilinx.com/products/intellectual-property/sem.html 70

[59] Xilinx, *Vivado Design Suite User Guide Partial Reconfiguration*, 2015, UG909 (v.2015.2). [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_3/ug909-vivado-partial-reconfiguration.pdf 71

[60] (2015) Zynq-7000 AP SoC Devices - Silicon Revision Differences. Xilinx. [Online]. Available: http://www.xilinx.com/support/answers/47916.html 99

[61] E. W. Blackmore, "Development of a large area neutron beam for system testing at TRIUMF," in *Radiation Effects Data Workshop*, 2009, pp. 20–24. 99

[62] K. Schoenberg, "The lansce accelerator: A powerful tool for science and applications," in *Particle Accelerator Conference, 2007. PAC. IEEE*, June 2007, pp. 120–120. 106

[63] Lansce. Neutron and Nuclear Science (WNR) Facility at LANSCE. [Online]. Available: http://wnr.lanl.gov/ 107

[64] C. Wilson, J. Stewart, P. Gauvin, J. MacKinnon, J. Coole, J. Uriste, A. George, G. Crum, E. Timmons, J. Beck, T. Flatley, M. Wirthlin, A. Wilson, and A. Stoddard, "CSP Hybrid Space Computing for STP-H5/ISEM on ISS," 2015. 110

[65] M. Wirthlin, M. Cannon, A. Camplani, M. Citterio, and C. Meroni, "Evaluating Multi-Gigabit Transceivers (MGT) for Use in High Energy Physics Through Proton Irradiation." in *2015 IEEE NUCLEAR AND SPACE RADIATION EFFECTS CONFERENCE*, July 2015. 111

[66] (2015) Ultrascale+ 16nm technology portfolio announcement and backgrounder. Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/backgrounders/backgrounder-16nm-ultrascale-plus-technology-and-portfolio-announcement.pdf 111

[67] Xilinx, *7 Series FPGAs Memory Resources*, 2014, UG473 (v.1.11). [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf 133

Appendix A

TMR Markov Modeling

The purpose of this Appendix is to motivate the importance of using scrubbing as a repair mechanism in coordination with TMR. The Markov models and equations to derive the reliability graphs and Mean Time to Failure (MTTF) calculations are given in [35].

The variable λ is the failure rate of the circuit based on how often upsets are expected to occur. The repair rate, μ, is defined as the frequency that the repair mechanism scans the entire memory. In practice, $\lambda \ll \mu$.

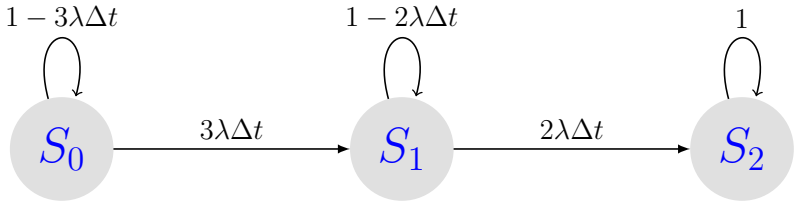

Figure A.1: Markov Model for a TMR System

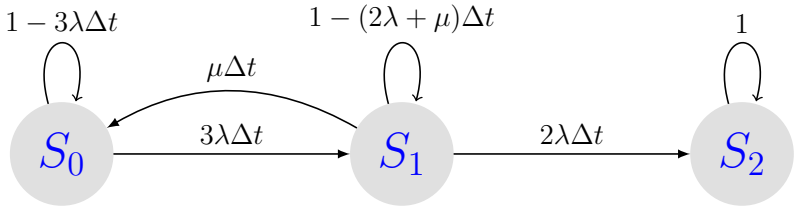

Figure A.2: Markov Model for a TMR System With Repair (TMRR)

It is useful to discuss the reliability models developed for TMR and TMR with repair (TMRR) when comparing improvements in reliability and Mean Time to Failure (MTTF). Figures A.1 and A.2 show the Markov models for TMR and TMRR. Solving the models for the respective reliability equations results in equations A.1, A.2a, and A.2b. These models assume a perfect voting mechanism since they do not factor in the probability of the voter failing in these equations.

$$R_{TMR}(t) = 3e^{-2\lambda t} - 2e^{-3\lambda t} \tag{A.1}$$

$$R_{TMRR}(t) = \frac{1}{\alpha_1 - \alpha_2}\left[(5\lambda + \mu) * (e^{\alpha_1 t} - e^{\alpha_2 t}) + \alpha_1 e^{\alpha_1 t} - \alpha_2 e^{\alpha_2 t}\right] \tag{A.2a}$$

$$\alpha_1, \alpha_2 = \frac{-(5\lambda + \mu) \pm \sqrt{\lambda^2 + \mu^2 + 10\lambda\mu}}{2} \tag{A.2b}$$

Figures A.3 and A.4 illustrate the difference of reliability over time when using TMR alone compared to using TMR with a repair mechanism. As the high curve in Figure A.4 shows, the reliability improvement is significant compared to the curve in Figure A.3. The failure rate used in these calculations is one upset every 10^2 hours. The failure rate in space may be slightly more frequent, but the trends shown in the graphs and table computed with this upset rate provide a realistic illustration.

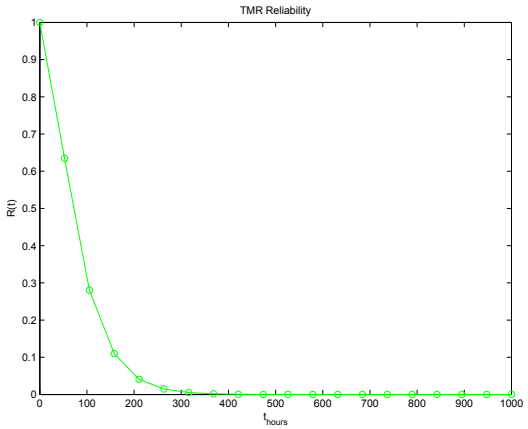

Figure A.3: Reliability of TMR for $\lambda = 10^{-2}$

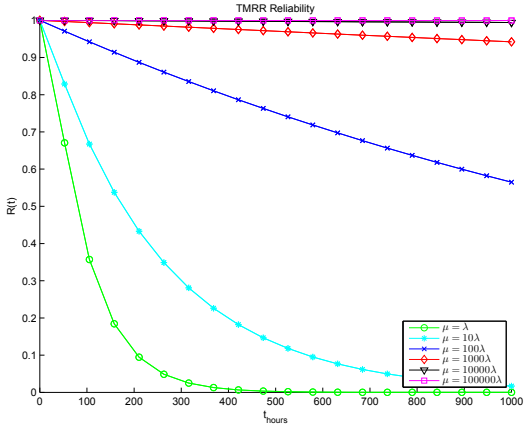

Figure A.4: Reliability of TMR With Repair for $\lambda = 10^{-2}$

In addition to reliability, the models presented here are given to compare the MTTFs of TMR with repair and TMR with no repair. The MTTF equations are equations A.3 and A.4 below:

$$MTTF_{TMR} = \frac{5}{6\lambda}, \tag{A.3}$$

and

$$MTTF_{TMRR} = \frac{5 + \frac{\mu}{\lambda}}{6\lambda}. \tag{A.4}$$

Table A.1 gives the resulting MTTFs for different repair rates μ. The last column shows the improvement gain when using a repair mechanism with TMR compared to using TMR alone. Notice how much of an improvement in MTTF is observed when increasing the frequency repair rate.

Table A.1: MTTF Comparison

$\lambda = 10^{-2}$ upsets/hour	**TMR**	**TMRR**	**Improvement over TMR**
$\mu = \lambda$	83.33	100	1.2
$\mu = 10\lambda$	83.33	250	3.0
$\mu = 100\lambda$	83.33	1750	21.0
$\mu = 1000\lambda$	83.33	16750	201.1
$\mu = 10000\lambda$	83.33	166750	2001.1
$\mu = 100000\lambda$	83.33	1666750	20001.8

Appendix B

PCAP Transfer Process

With the hardware infrastructure in place, the Zynq SoC can host its own scrubber without requiring a large hardware design to be implemented in its configuration PL or in an external chip that connects via cable to an interface on the Zynq device board. The processors run software code that resides in DDR memory and performs scrubbing operations including readback, writes to the configuration memory, or accesses to the internal configuration registers in the PL through the DevC and PCAP interfaces (see Chapter 7).

The scrubbing program (.elf file) can be generated by the Xilinx SDK. This .elf file can be programmed into on-board flash memory, loaded onto an SD card from which the Zynq can boot, or be initiated over JTAG using the Xilinx SDK or iMPACT tools.

To perform data transfers through the PCAP and DevC interfaces, the device must first be configured and initialized. The procedure to initialize the PL is as follows:

1. Unlock the DevC interface with the UNLOCK key.
2. Wait for the PL to receive power (PCFG_POR_B).
3. Wait for the PL to be initialized.
4. Enable the PCAP bridge and select PCAP for reconfiguration.
5. Make sure the initial PL configuration is done.

After configuring and initializing the PL, configuration command sequences may be sent to the PL using the commands below. Although it may seem redundant to perform some steps repeatedly with multiple command sequences, it is good practice to follow the same sequence of steps to ensure that certain options have not been disabled inadvertently by other applications. The DevC interface greatly simplifies configuration operations down to basic DMA transfers where valid sources, destinations, and lengths are the only operands necessary.

1. Disable the PCAP loopback.
2. Set the data clock rate to be every clock cycle.
3. Queue-up a DMA transfer using the devcfg DMA registers:
   (a) Source Address: Location of new PL bitstream.
   (b) Destination Address: 0xFFFF_FFFF (PL address).

(c) Source Length: Total number of 32-bit words in the new PL bitstream.

(d) Destination Length: Total number of 32-bit words in the new PL bitstream. A write to this register last moves the values of all four registers into the Command Queue.

4. Wait for the DMA and PCAP transfers to finish.

When reading from or writing to the configuration logic, a special address code is used to specify the PL or configuration logic as the DMA destination (or source in the case of reading). Writes to the configuration logic differ from reads in that writes only require one DMA transfer, (namely to send the data to the configuration module to be written). Reads require two transfers: the first to issue the read command, and the second to receive the data sent back by the configuration module.

Appendix C

Proprietary CRC Registers

The material in this appendix is proprietary to Xilinx Inc. and cannot be published in this public document.

Appendix D

Configuration Command Sequences for the PCAP

The configuration command sequences given in [1] are specific to the JTAG and SelectMAP configuration interfaces. While some of the commands in each sequence are required regardless of the interface, less important commands such as NO-OPs and Dummy words may be necessary in variable quantities depending on the configuration interface. Oftentimes, these less important commands are present merely to flush buffers or give extra time to the configuration logic for processing previous commands.

This appendix presents the bare minimum sequences for read and write operations based on experimentation using the PCAP interface. The following sections are divided into write and read operations. They are further subdivided between configuration register accesses and configuration memory accesses.

D.1 Write Operations

This section gives the command sequences for writing to configuration registers and configuration data. The first hex digits of each command word specify whether it is a write or read command.

D.1.1 Write Configuration Registers

This section presents the sequence to write to a configuration register, followed by the command codes for registers commonly used in scrubbing. Oftentimes, when writing to the configuration data (see next section), writes to configuration registers must be performed immediately preceding the configuration data.

Table D.1: Write Configuration Register Sequence

Command Word	Name
0xFFFFFFFF	Dummy
0xAA995566	Sync Word
0x20000000	NO-OP
0x300XX001	Write Config Reg
0xXXXXXXXX	Reg Value
0x30008001	Write CMD Register
0x0000000D	DESYNC

Table D.1 shows the command sequence to write to a configuration register. Remember that when writing to any configuration register, a value must always immediately follow the "write register" command. These values will naturally differ depending on the register being written to, but a space must always be reserved directly after every register-write command for the value.

Table D.2: Write Configuration Registers Commands

Command Word	Name
0x30026001	RBCRC_SW
0x3001C001	COR1
0x30002001	FAR
0x30004000	FDRI
0x3000C001	MASK
0x3000A001	CTL0
0x30008001	CMD
0x30018001	IDCODE

Table D.2 shows the specific command words for writing to configuration registers. Writing to the FAR is unusual in that if a DESYNC is issued at the end of its sequence and the Readback CRC is enabled, the Readback CRC will gain configuration logic access and start reading back frames, thus changing the FAR. So if the Readback CRC is used, do not append the DESYNC command to the end of the "write FAR" sequence.

D.1.2 Write Configuration Data

Table D.3 shows the full command sequence to write to the configuration memory. Recall that a dummy frame must be appended after all of the configuration data to flush the frame buffer so that all of the data emerges from the configruation memory.

Writing to the configuration data has a few unique characteristics. It is required to write to the IDCODE register with the actual ID of the FPGA before being able to write to the configuration. It is also required to write to the CMD register with the write configuration command (WCFG). Typically, it is a wise practice to include a write to the FAR in the sequence with the value of the starting address just in case the FAR changed since the last operation.

D.2 Read Operations

The most important difference between the reads and writes with the PCAP is that the write operations are always one transfer while the read operations require two transfers. The DESYNC command is necessarily appended at the end of every write sequence in order to release the configuration logic and allow it to be accessed by the Readback CRC hardware (if using the hybrid architecture).

Table D.3: Write Configuration Command Sequence

Command Word	Name
0xFFFFFFFF	Dummy
0xAA995566	Sync Word
0x20000000	NO-OP
0x30018001	Write IDCODE
0xXXXXXXXX	Device ID
0x30002001	Write FAR
0xXXXXXXXX	FRAD
0x30008001	Write CMD Register
0x00000001	WCFG
0x30004000	Write FDRI
0x5XXXXXXX	Data Word Count
0xXXXXXXXX	Data Word 0
...	Data Words 1 to n-2
0xXXXXXXXX	Data Word n-1
0x00000000	Dummy Word 0
...	Dummy Word 1 to 99
0x00000000	Dummy Word 100
0x30008001	Write CMD Register
0x0000000D	DESYNC

Appending the DESYNC command to the end of a PCAP readback sequence is futile because the data must be read via the second transfer before a DESYNC will have any effect. Sending the DESYNC as a third transfer, after receiving the readback data in the second transfer, will correctly be accepted and interpreted to release the configuration module. This explanation is the reason that none of the read operations have DESYNCs at the end of their sequences.

D.2.1 Read Configuration Registers

Reading Configuration Registers is one of the simplest operations. Table D.4 gives the command sequence to read a configuration register. Again, for the PCAP interface no DESYNCs are issued because the second transfer from the configuration module to the PCAP must occur before a DESYNC will have any effect. The second transfer returns the 32-bit value of the specified configuration register.

Table D.5 gives the command words for the configuration registers most commonly read by scrubbers. Many of these registers, including RBCRC_LIVE and STATUS, are read-only registers.

Table D.4: Read Configuration Register Sequence

Command Word	Name
0xAA995566	Sync Word
0x20000000	NO-OP
0x280XX001	Read Config Reg

Table D.5: Read Configuration Registers Commands

Command Word	Name
0x28026001	RBCRC_SW
0x28024001	RBCRC_HW
0x28028001	RBCRC_LIVE
0x28002001	FAR
0x28006000	FDRO
0x2800E001	STATUS
0x2800A001	CTL0
0x2801C001	COR1
0x28018001	IDCODE

D.2.2 Performing Readback of Configuration Data

Table D.6 gives the command sequence to perform a readback of configuration data. Unlike the write configuration sequence, the readback command sequence does not require writing to the IDCODE register. Writing to the FAR is optional in this sequence. It is often a good practice to keep the write to the FAR and the readback operations separate in case there is a need to debug whether the FAR is getting written with the correct value. The data word count must include an extra dummy frame's length in its value to account for the frame buffer otherwise the PL to PCAP transfer will just return a frame of zeros.

Table D.6: Readback Command Sequence

Command Word	Name
0xAA995566	Sync Word
0x30008001	Write CMD Register
0x00000004	RCFG
0x20000000	NO-OP
0x28006000	Read FDRO
0x48XXXXXX	Data Word Count

Appendix E

Scrubbing Other Memory Types

Almost all of the scrubbers mentioned in this thesis exclusively scrub static configuration data. These types of scrubbers rely on the assumption that configuration data should always maintain its original state. Any changes are deemed to be upsets. Since changeable memories do exist in FPGAs, their reliability must be protected via other mechanisms apart from configuration scrubbers. This appendix gives examples of how similar mitigation techniques can monitor and scrub the changeable memories in an FPGA including BRAMs, CLB registers and latches, LUTRAMs, SRLs, and Functional Blocks with Dynamic Reconfiguration Ports (DRPs).

E.1 Block RAM ECC

The next largest physical allotment of memory bits after configuration bits, are Block RAMs (BRAMs). BRAMs are large dual-ported memory blocks that prove useful in many user applications. Because they are writable memories, BRAM contents often change, making it difficult to detect when upsets occur inside them using the scrubbing approaches described in this thesis. Thus, other techniques for preserving reliability must be adapted for BRAMs.

Xilinx provides a unique ECC protection for the BRAMs in their FPGAs [67]. The ECC is a few extra bits (depending on the width of the data word) stored with the data word. For instance a 64-bit word has 8 parity bits for a total storage footprint of 72 bits. Whenever a write operation takes place, the ECC bits are recalculated and stored along with the data word. A read operation triggers a new ECC to be calculated and compared to the stored value. This ECC mechanism achieves a similar SECDED capability as does the ECC for configuration scrubbing. If the BRAM is dual-ported, and the read operation detects an ECC mismatch, the upset bit can be corrected immediately.

Another upset mitigation mechanism for BRAMs was developed in [15]. The BRAM scrubber was a TMR design that triplicated the instruction memory and sequentially counted through each of the addresses to read out the data and compare the three copies. Whenever a discrepancy was observed, the scrubber wrote the majority value into the memory segment with the difference. The BRAM memory was dual-ported which allowed this BRAM scrubber to monitor these internal memories as well as perform scrubbing operations on the actual configuration memory.

E.2 Dynamic Reconfiguration Ports

Functional block memories, such as clock management tiles, XADC, and serial transceivers, are a subset of configuration memory bits that can be enabled by a user design [1]. Although these blocks usually maintain their initial values throughout the life of the user application, they sometimes will require a change. These blocks are unique, however, in that when they are enabled, the address space of the functional block does not align perfectly to the configuration frame boundaries. While partial reconfiguration is one option to scrub the frame, doing so would most likely completely reset the functional block, which could create larger problems.

A solution is to use the dedicated mini-configuration ports within these blocks called Dynamic Reconfiguration Ports (DRPs). DRPs handle writes and reads to the block itself that respect the unique block-addressing layout. The frames in these memories need not be reset by rewriting all of the configuration bits via a partial reconfiguration scrub.

DRPs, similar to the ICAP interface, require an instantiated primitive in the user design. A controller circuit must also be implemented to connect into and manage the DRP. The scrubbing would be accomplished by monitoring only the functional block memory bits with values associated with the current state of the block. By respecting the address boundaries, the DRPs can effectively scrub the data that the functional blocks should contain or easily replace it when necessary, while the functional block remains operational.

E.3 Readback Capture

A useful feature for reading the values of changeable memory locations, that has existed since the launch of the Virtex-4 family, is known as Readback Capture. When implemented, this feature loads the current states of BRAM, SRL, and CLB registers into configuration bits [1]. Followed by normal readback, all of these values are read like normal configuration bits. Using the captured values as a stable restore point is one type of potential mitigation recovery strategy that could be used for these types of memories when upsets occur. One might conceive of how using such a mechanism could be feasible to create a scrubber for these types of memories.

Appendix F

FRADs List for Zynq XZC07020 FPGA

This appendix gives excerpts from a sample FRADs list for the Zynq XZC07020 7-Series FPGA. Not all of the FRADs are present, but notice key boundary transitions between all rows.

0: FRAD: 0x0000000, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=0
1: FRAD: 0x0000001, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=1
2: FRAD: 0x0000002, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=2
3: FRAD: 0x0000003, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=3
4: FRAD: 0x0000004, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=4
5: FRAD: 0x0000005, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=5
6: FRAD: 0x0000006, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=6
7: FRAD: 0x0000007, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=7
8: FRAD: 0x0000008, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=8
9: FRAD: 0x0000009, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=9
10: FRAD: 0x000000A, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=10
11: FRAD: 0x000000B, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=11
12: FRAD: 0x000000C, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=12
13: FRAD: 0x000000D, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=13
14: FRAD: 0x000000E, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=14
15: FRAD: 0x000000F, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=15
16: FRAD: 0x0000010, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=16
17: FRAD: 0x0000011, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=17
18: FRAD: 0x0000012, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=18
19: FRAD: 0x0000013, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=19
20: FRAD: 0x0000014, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=20
21: FRAD: 0x0000015, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=21
22: FRAD: 0x0000016, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=22
23: FRAD: 0x0000017, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=23
24: FRAD: 0x0000018, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=24
25: FRAD: 0x0000019, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=25
26: FRAD: 0x000001A, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=26
27: FRAD: 0x000001B, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=27
28: FRAD: 0x000001C, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=28
29: FRAD: 0x000001D, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=29
30: FRAD: 0x000001E, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=30
31: FRAD: 0x000001F, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=31
32: FRAD: 0x0000020, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=32

33: FRAD: 0x0000021, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=33
34: FRAD: 0x0000022, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=34
35: FRAD: 0x0000023, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=35
36: FRAD: 0x0000024, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=36
37: FRAD: 0x0000025, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=37
38: FRAD: 0x0000026, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=38
39: FRAD: 0x0000027, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=39
40: FRAD: 0x0000028, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=40
41: FRAD: 0x0000029, Type=LOGIC(0), Half=top, Row=0, Column=0, Minor=41
42: FRAD: 0x0000080, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=0
43: FRAD: 0x0000081, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=1
44: FRAD: 0x0000082, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=2
45: FRAD: 0x0000083, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=3
46: FRAD: 0x0000084, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=4
47: FRAD: 0x0000085, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=5
48: FRAD: 0x0000086, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=6
49: FRAD: 0x0000087, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=7
50: FRAD: 0x0000088, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=8
51: FRAD: 0x0000089, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=9
52: FRAD: 0x000008A, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=10
53: FRAD: 0x000008B, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=11
54: FRAD: 0x000008C, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=12
55: FRAD: 0x000008D, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=13
56: FRAD: 0x000008E, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=14
57: FRAD: 0x000008F, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=15
58: FRAD: 0x0000090, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=16
59: FRAD: 0x0000091, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=17
60: FRAD: 0x0000092, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=18
61: FRAD: 0x0000093, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=19
62: FRAD: 0x0000094, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=20
63: FRAD: 0x0000095, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=21
64: FRAD: 0x0000096, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=22
65: FRAD: 0x0000097, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=23
66: FRAD: 0x0000098, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=24
67: FRAD: 0x0000099, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=25
68: FRAD: 0x000009A, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=26
69: FRAD: 0x000009B, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=27
70: FRAD: 0x000009C, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=28
71: FRAD: 0x000009D, Type=LOGIC(0), Half=top, Row=0, Column=1, Minor=29
72: FRAD: 0x0000100, Type=LOGIC(0), Half=top, Row=0, Column=2, Minor=0
73: FRAD: 0x0000101, Type=LOGIC(0), Half=top, Row=0, Column=2, Minor=1
74: FRAD: 0x0000102, Type=LOGIC(0), Half=top, Row=0, Column=2, Minor=2
...
2558: FRAD: 0x00024A4, Type=LOGIC(0), Half=top, Row=0, Column=73, Minor=36
2559: FRAD: 0x00024A5, Type=LOGIC(0), Half=top, Row=0, Column=73, Minor=37
2560: FRAD: 0x00024A6, Type=LOGIC(0), Half=top, Row=0, Column=73, Minor=38
2561: FRAD: 0x00024A7, Type=LOGIC(0), Half=top, Row=0, Column=73, Minor=39

2562: FRAD: 0x00024A8, Type=LOGIC(0), Half=top, Row=0, Column=73, Minor=40
2563: FRAD: 0x00024A9, Type=LOGIC(0), Half=top, Row=0, Column=73, Minor=41
2564: FRAD: 0x0400000, Type=LOGIC(0), Half=bottom, Row=0, Column=0, Minor=0
2565: FRAD: 0x0400001, Type=LOGIC(0), Half=bottom, Row=0, Column=0, Minor=1
2566: FRAD: 0x0400002, Type=LOGIC(0), Half=bottom, Row=0, Column=0, Minor=2
...
5125: FRAD: 0x04024A7, Type=LOGIC(0), Half=bottom, Row=0, Column=73, Minor=39
5126: FRAD: 0x04024A8, Type=LOGIC(0), Half=bottom, Row=0, Column=73, Minor=40
5127: FRAD: 0x04024A9, Type=LOGIC(0), Half=bottom, Row=0, Column=73, Minor=41
5128: FRAD: 0x0420000, Type=LOGIC(0), Half=bottom, Row=1, Column=0, Minor=0
5129: FRAD: 0x0420001, Type=LOGIC(0), Half=bottom, Row=1, Column=0, Minor=1
5130: FRAD: 0x0420002, Type=LOGIC(0), Half=bottom, Row=1, Column=0, Minor=2
...
5167: FRAD: 0x0420027, Type=LOGIC(0), Half=bottom, Row=1, Column=0, Minor=39
5168: FRAD: 0x0420028, Type=LOGIC(0), Half=bottom, Row=1, Column=0, Minor=40
5169: FRAD: 0x0420029, Type=LOGIC(0), Half=bottom, Row=1, Column=0, Minor=41
5170: FRAD: 0x0420080, Type=LOGIC(0), Half=bottom, Row=1, Column=1, Minor=0
5171: FRAD: 0x0420081, Type=LOGIC(0), Half=bottom, Row=1, Column=1, Minor=1
5172: FRAD: 0x0420082, Type=LOGIC(0), Half=bottom, Row=1, Column=1, Minor=2
...
7647: FRAD: 0x042241B, Type=LOGIC(0), Half=bottom, Row=1, Column=72, Minor=27
7648: FRAD: 0x042241C, Type=LOGIC(0), Half=bottom, Row=1, Column=72, Minor=28
7649: FRAD: 0x042241D, Type=LOGIC(0), Half=bottom, Row=1, Column=72, Minor=29
7650: FRAD: 0x0422480, Type=LOGIC(0), Half=bottom, Row=1, Column=73, Minor=0
7651: FRAD: 0x0422481, Type=LOGIC(0), Half=bottom, Row=1, Column=73, Minor=1
7652: FRAD: 0x0422482, Type=LOGIC(0), Half=bottom, Row=1, Column=73, Minor=2
...
7689: FRAD: 0x04224A7, Type=LOGIC(0), Half=bottom, Row=1, Column=73, Minor=39
7690: FRAD: 0x04224A8, Type=LOGIC(0), Half=bottom, Row=1, Column=73, Minor=40
7691: FRAD: 0x04224A9, Type=LOGIC(0), Half=bottom, Row=1, Column=73, Minor=41
7692: FRAD: 0x0800000, Type=BRAM(1), Half=top, Row=0, Column=0, Minor=0
7693: FRAD: 0x0800001, Type=BRAM(1), Half=top, Row=0, Column=0, Minor=1
7694: FRAD: 0x0800002, Type=BRAM(1), Half=top, Row=0, Column=0, Minor=2
...
7817: FRAD: 0x080007D, Type=BRAM(1), Half=top, Row=0, Column=0, Minor=125
7818: FRAD: 0x080007E, Type=BRAM(1), Half=top, Row=0, Column=0, Minor=126
7819: FRAD: 0x080007F, Type=BRAM(1), Half=top, Row=0, Column=0, Minor=127
7820: FRAD: 0x0800080, Type=BRAM(1), Half=top, Row=0, Column=1, Minor=0
7821: FRAD: 0x0800081, Type=BRAM(1), Half=top, Row=0, Column=1, Minor=1
7822: FRAD: 0x0800082, Type=BRAM(1), Half=top, Row=0, Column=1, Minor=2
...
8329: FRAD: 0x080027D, Type=BRAM(1), Half=top, Row=0, Column=4, Minor=125
8330: FRAD: 0x080027E, Type=BRAM(1), Half=top, Row=0, Column=4, Minor=126
8331: FRAD: 0x080027F, Type=BRAM(1), Half=top, Row=0, Column=4, Minor=127
8332: FRAD: 0x0800280, Type=BRAM(1), Half=top, Row=0, Column=5, Minor=0
8333: FRAD: 0x0800281, Type=BRAM(1), Half=top, Row=0, Column=5, Minor=1
8334: FRAD: 0x0800282, Type=BRAM(1), Half=top, Row=0, Column=5, Minor=2

...
8457: FRAD: 0x08002FD, Type=BRAM(1), Half=top, Row=0, Column=5, Minor=125
8458: FRAD: 0x08002FE, Type=BRAM(1), Half=top, Row=0, Column=5, Minor=126
8459: FRAD: 0x08002FF, Type=BRAM(1), Half=top, Row=0, Column=5, Minor=127
8460: FRAD: 0x0C00000, Type=BRAM(1), Half=bottom, Row=0, Column=0, Minor=0
8461: FRAD: 0x0C00001, Type=BRAM(1), Half=bottom, Row=0, Column=0, Minor=1
8462: FRAD: 0x0C00002, Type=BRAM(1), Half=bottom, Row=0, Column=0, Minor=2
...
8585: FRAD: 0x0C0007D, Type=BRAM(1), Half=bottom, Row=0, Column=0, Minor=125
8586: FRAD: 0x0C0007E, Type=BRAM(1), Half=bottom, Row=0, Column=0, Minor=126
8587: FRAD: 0x0C0007F, Type=BRAM(1), Half=bottom, Row=0, Column=0, Minor=127
8588: FRAD: 0x0C00080, Type=BRAM(1), Half=bottom, Row=0, Column=1, Minor=0
8589: FRAD: 0x0C00081, Type=BRAM(1), Half=bottom, Row=0, Column=1, Minor=1
8590: FRAD: 0x0C00082, Type=BRAM(1), Half=bottom, Row=0, Column=1, Minor=2
...
9097: FRAD: 0x0C0027D, Type=BRAM(1), Half=bottom, Row=0, Column=4, Minor=125
9098: FRAD: 0x0C0027E, Type=BRAM(1), Half=bottom, Row=0, Column=4, Minor=126
9099: FRAD: 0x0C0027F, Type=BRAM(1), Half=bottom, Row=0, Column=4, Minor=127
9100: FRAD: 0x0C00280, Type=BRAM(1), Half=bottom, Row=0, Column=5, Minor=0
9101: FRAD: 0x0C00281, Type=BRAM(1), Half=bottom, Row=0, Column=5, Minor=1
9102: FRAD: 0x0C00282, Type=BRAM(1), Half=bottom, Row=0, Column=5, Minor=2
...
9225: FRAD: 0x0C002FD, Type=BRAM(1), Half=bottom, Row=0, Column=5, Minor=125
9226: FRAD: 0x0C002FE, Type=BRAM(1), Half=bottom, Row=0, Column=5, Minor=126
9227: FRAD: 0x0C002FF, Type=BRAM(1), Half=bottom, Row=0, Column=5, Minor=127
9228: FRAD: 0x0C20000, Type=BRAM(1), Half=bottom, Row=1, Column=0, Minor=0
9229: FRAD: 0x0C20001, Type=BRAM(1), Half=bottom, Row=1, Column=0, Minor=1
9230: FRAD: 0x0C20002, Type=BRAM(1), Half=bottom, Row=1, Column=0, Minor=2
...
9353: FRAD: 0x0C2007D, Type=BRAM(1), Half=bottom, Row=1, Column=0, Minor=125
9354: FRAD: 0x0C2007E, Type=BRAM(1), Half=bottom, Row=1, Column=0, Minor=126
9355: FRAD: 0x0C2007F, Type=BRAM(1), Half=bottom, Row=1, Column=0, Minor=127
9356: FRAD: 0x0C20080, Type=BRAM(1), Half=bottom, Row=1, Column=1, Minor=0
9357: FRAD: 0x0C20081, Type=BRAM(1), Half=bottom, Row=1, Column=1, Minor=1
9358: FRAD: 0x0C20082, Type=BRAM(1), Half=bottom, Row=1, Column=1, Minor=2
...
9865: FRAD: 0x0C2027D, Type=BRAM(1), Half=bottom, Row=1, Column=4, Minor=125
9866: FRAD: 0x0C2027E, Type=BRAM(1), Half=bottom, Row=1, Column=4, Minor=126
9867: FRAD: 0x0C2027F, Type=BRAM(1), Half=bottom, Row=1, Column=4, Minor=127
9868: FRAD: 0x0C20280, Type=BRAM(1), Half=bottom, Row=1, Column=5, Minor=0
9869: FRAD: 0x0C20281, Type=BRAM(1), Half=bottom, Row=1, Column=5, Minor=1
9870: FRAD: 0x0C20282, Type=BRAM(1), Half=bottom, Row=1, Column=5, Minor=2
...
9993: FRAD: 0x0C202FD, Type=BRAM(1), Half=bottom, Row=1, Column=5, Minor=125
9994: FRAD: 0x0C202FE, Type=BRAM(1), Half=bottom, Row=1, Column=5, Minor=126
9995: FRAD: 0x0C202FF, Type=BRAM(1), Half=bottom, Row=1, Column=5, Minor=127
9996: FRAD: 0x1000000, Type=CFG_CLB(2), Half=top, Row=0, Column=0, Minor=0

9997: FRAD: 0x1000080, Type=CFG_CLB(2), Half=top, Row=0, Column=1, Minor=0
9998: FRAD: 0x1000100, Type=CFG_CLB(2), Half=top, Row=0, Column=2, Minor=0
...
10067: FRAD: 0x1002380, Type=CFG_CLB(2), Half=top, Row=0, Column=71, Minor=0
10068: FRAD: 0x1002400, Type=CFG_CLB(2), Half=top, Row=0, Column=72, Minor=0
10069: FRAD: 0x1002480, Type=CFG_CLB(2), Half=top, Row=0, Column=73, Minor=0
10070: FRAD: 0x1400000, Type=CFG_CLB(2), Half=bottom, Row=0, Column=0, Minor=0
10071: FRAD: 0x1400080, Type=CFG_CLB(2), Half=bottom, Row=0, Column=1, Minor=0
10072: FRAD: 0x1400100, Type=CFG_CLB(2), Half=bottom, Row=0, Column=2, Minor=0
...
10141: FRAD: 0x1402380, Type=CFG_CLB(2), Half=bottom, Row=0, Column=71, Minor=0
10142: FRAD: 0x1402400, Type=CFG_CLB(2), Half=bottom, Row=0, Column=72, Minor=0
10143: FRAD: 0x1402480, Type=CFG_CLB(2), Half=bottom, Row=0, Column=73, Minor=0
10144: FRAD: 0x1420000, Type=CFG_CLB(2), Half=bottom, Row=1, Column=0, Minor=0
10145: FRAD: 0x1420080, Type=CFG_CLB(2), Half=bottom, Row=1, Column=1, Minor=0
10146: FRAD: 0x1420100, Type=CFG_CLB(2), Half=bottom, Row=1, Column=2, Minor=0
...
10215: FRAD: 0x1422380, Type=CFG_CLB(2), Half=bottom, Row=1, Column=71, Minor=0
10216: FRAD: 0x1422400, Type=CFG_CLB(2), Half=bottom, Row=1, Column=72, Minor=0
10217: FRAD: 0x1422480, Type=CFG_CLB(2), Half=bottom, Row=1, Column=73, Minor=0
10218: FRAD: 0x1800000, Type=UNKNOWN(3), Half=top, Row=0, Column=0, Minor=0
10219: FRAD: 0x1800080, Type=UNKNOWN(3), Half=top, Row=0, Column=1, Minor=0
10220: FRAD: 0x1800100, Type=UNKNOWN(3), Half=top, Row=0, Column=2, Minor=0
10221: FRAD: 0x1800180, Type=UNKNOWN(3), Half=top, Row=0, Column=3, Minor=0
10222: FRAD: 0x1800200, Type=UNKNOWN(3), Half=top, Row=0, Column=4, Minor=0
10223: FRAD: 0x1800280, Type=UNKNOWN(3), Half=top, Row=0, Column=5, Minor=0
10224: FRAD: 0x1C00000, Type=UNKNOWN(3), Half=bottom, Row=0, Column=0, Minor=0
10225: FRAD: 0x1C00080, Type=UNKNOWN(3), Half=bottom, Row=0, Column=1, Minor=0
10226: FRAD: 0x1C00100, Type=UNKNOWN(3), Half=bottom, Row=0, Column=2, Minor=0
10227: FRAD: 0x1C00180, Type=UNKNOWN(3), Half=bottom, Row=0, Column=3, Minor=0
10228: FRAD: 0x1C00200, Type=UNKNOWN(3), Half=bottom, Row=0, Column=4, Minor=0
10229: FRAD: 0x1C00280, Type=UNKNOWN(3), Half=bottom, Row=0, Column=5, Minor=0
10230: FRAD: 0x1C20000, Type=UNKNOWN(3), Half=bottom, Row=1, Column=0, Minor=0
10231: FRAD: 0x1C20080, Type=UNKNOWN(3), Half=bottom, Row=1, Column=1, Minor=0
10232: FRAD: 0x1C20100, Type=UNKNOWN(3), Half=bottom, Row=1, Column=2, Minor=0
10233: FRAD: 0x1C20180, Type=UNKNOWN(3), Half=bottom, Row=1, Column=3, Minor=0
10234: FRAD: 0x1C20200, Type=UNKNOWN(3), Half=bottom, Row=1, Column=4, Minor=0
10235: FRAD: 0x1C20280, Type=UNKNOWN(3), Half=bottom, Row=1, Column=5, Minor=0
10236: FRAD: 0x2000000, Type=UNKNOWN(4), Half=top, Row=0, Column=0, Minor=0
10237: FRAD: 0x2000080, Type=UNKNOWN(4), Half=top, Row=0, Column=1, Minor=0
10238: FRAD: 0x2000100, Type=UNKNOWN(4), Half=top, Row=0, Column=2, Minor=0
10239: FRAD: 0x2000180, Type=UNKNOWN(4), Half=top, Row=0, Column=3, Minor=0
...
10240: FRAD: 0x2000200, Type=UNKNOWN(4), Half=top, Row=0, Column=4, Minor=0
10241: FRAD: 0x2000280, Type=UNKNOWN(4), Half=top, Row=0, Column=5, Minor=0
10242: FRAD: 0x2000300, Type=UNKNOWN(4), Half=top, Row=0, Column=6, Minor=0
10243: FRAD: 0x2000380, Type=UNKNOWN(4), Half=top, Row=0, Column=7, Minor=0

10244: FRAD: 0x2000400, Type=UNKNOWN(4), Half=top, Row=0, Column=8, Minor=0
10245: FRAD: 0x2000480, Type=UNKNOWN(4), Half=top, Row=0, Column=9, Minor=0
10246: FRAD: 0x2000500, Type=UNKNOWN(4), Half=top, Row=0, Column=10, Minor=0
10247: FRAD: 0x2000580, Type=UNKNOWN(4), Half=top, Row=0, Column=11, Minor=0
10248: FRAD: 0x2000600, Type=UNKNOWN(4), Half=top, Row=0, Column=12, Minor=0
10249: FRAD: 0x2000680, Type=UNKNOWN(4), Half=top, Row=0, Column=13, Minor=0
10250: FRAD: 0x2000700, Type=UNKNOWN(4), Half=top, Row=0, Column=14, Minor=0
10251: FRAD: 0x2000780, Type=UNKNOWN(4), Half=top, Row=0, Column=15, Minor=0
10252: FRAD: 0x2000800, Type=UNKNOWN(4), Half=top, Row=0, Column=16, Minor=0
10253: FRAD: 0x2000880, Type=UNKNOWN(4), Half=top, Row=0, Column=17, Minor=0
10254: FRAD: 0x2000900, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=0
10255: FRAD: 0x2000901, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=1
10256: FRAD: 0x2000902, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=2
10257: FRAD: 0x2000903, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=3
...
10366: FRAD: 0x2000970, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=112
10367: FRAD: 0x2000971, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=113
10368: FRAD: 0x2000972, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=114
10369: FRAD: 0x2000973, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=115
10370: FRAD: 0x2000974, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=116
10371: FRAD: 0x2000975, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=117
10372: FRAD: 0x2000976, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=118
10373: FRAD: 0x2000977, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=119
10374: FRAD: 0x2000978, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=120
10375: FRAD: 0x2000979, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=121
10376: FRAD: 0x200097A, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=122
10377: FRAD: 0x200097B, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=123
10378: FRAD: 0x200097C, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=124
10379: FRAD: 0x200097D, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=125
10380: FRAD: 0x200097E, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=126
10381: FRAD: 0x200097F, Type=UNKNOWN(4), Half=top, Row=0, Column=18, Minor=127

Appendix G

Excerpts from Hybrid Scrubber Log

This appendix gives excerpts from a hybrid scrubber radiation test. The excerpts include different examples of upset types (refer to Sections 4.3.2 and 4.3.3) and shows the detection and correction messages. In the last excerpt an example of a full readback scrub is given which is nearly identical to the standard output of a readback scrubber.

Note that many of the MBUs were caused by a single particle strike. Due to the interleaving of the frames (see Section 4.1.3), many of the upsets occur by one particle strike affecting adjacent frames [48]. This first excerpt presents a simple SBU.

```
Frame_ECC 0: 870F1527
Frame_ECC 1: 60001522
Frame_ECC 2: CRC = 1
Frame_ECC 3: 2
Frame_ECC 7: 8

Polling for 0 iterations, FIFO_COUNT: 2

129.481941 : SBU corrected @
        FRAD: 1522 Word: 15 Bit: 7
**************************************************************
```

The next log output conveys an even-numbered → even-numbered upset.

```
Frame_ECC 0: 80000001
Frame_ECC 1: 20421B9C
Frame_ECC 2: CRC = 1
Frame_ECC 3: 4
Frame_ECC 7: 10
Polling for 0 iterations, FIFO_COUNT: 4

279.021899 : CRC HIGH: Multi-Bit Upset! Syndrome: 1

279.022379 : Scrubbing FRAD: 421B9C...
```

```
279.022460 : FAULT DETECTED! FRAD: 421B9C Word: 9 Bit(s): 14 15
279.022489 : WORD HAD MULTIPLE BITS UPSET WITH 2 ERRONEOUS BITS
279.022514 : Word: 9 | Expected: 0 Actual: C000

279.022609 : Scrubbing of FRAD: 421B9C Finished with 2 Upsets !

Clearing CRC latch !
After CRC Scrub, CRC = 1
************************************************************
Polling for 0 iterations, FIFO_COUNT: 4

279.199548 : CRC HIGH: Multi-Bit Upset! Syndrome: 3

279.199933 : Scrubbing FRAD: 421B9D...
279.199988 : FAULT DETECTED! FRAD: 421B9D Word: 9 Bit(s): 13 14
279.200013 : WORD HAD MULTIPLE BITS UPSET WITH 2 ERRONEOUS BITS
279.200037 : Word: 9 | Expected: 0 Actual: 6000

279.200128 : Scrubbing of FRAD: 421B9D Finished with 2 Upsets !

Clearing CRC latch !
After CRC Scrub, CRC = 0
************************************************************
```

The next excerpt presents an example of an even-numbered upset followed by an odd-numbered upset (in this case an SBU).

```
Frame_ECC 0: 80000007
Frame_ECC 1: 20421A1A
Frame_ECC 2: CRC = 1
Frame_ECC 3: 3
Frame_ECC 7: 18
Polling for 0 iterations, FIFO_COUNT: 3

206.937119 : CRC HIGH: Multi-Bit Upset! Syndrome: 7

206.937603 : Scrubbing FRAD: 421A1A...
206.937687 : FAULT DETECTED! FRAD: 421A1A Word: 0 Bit(s): 0
206.937716 : Word: 0 | Expected: 0 Actual: 1
206.937754 : FAULT DETECTED! FRAD: 421A1A Word: 83 Bit(s): 27 28
206.937780 : WORD HAD MULTIPLE BITS UPSET WITH 2 ERRONEOUS BITS
206.937805 : Word: 83 | Expected: 0 Actual: 18000000
```

```
206.937892 : Scrubbing of FRAD: 421A1A Finished with 3 Upsets !

Clearing CRC latch !

206.973279 : ACCIDENTAL CORRECTION:
        FRAD: 421A1A Word: 0 Bit: 0 Syndrome: 1DDC !
207.043996 : ACCIDENTAL CORRECTION:
        FRAD: 421A1A Word: 0 Bit: 7 Syndrome: 1327 !

After CRC Scrub, CRC = 0
**************************************************************
Polling for 0 iterations, FIFO_COUNT: 2
Peek at Next Syndrome: 0 CRC = 0

207.115024 : SBU corrected by Readback CRC at
        FRAD: 421A1B Word: 83 Bit: 28 !
**************************************************************
```

This excerpt gives a triple-bit upset followed by a double-bit upset in the next frame.

```
Frame_ECC 0: 9E19167E
Frame_ECC 1: 60421B1C
Frame_ECC 2: CRC = 1
Frame_ECC 3: 2
Frame_ECC 7: 36
Polling for 0 iterations, FIFO_COUNT: 2

511.318261 : CRC HIGH: Multi-Bit Upset! Syndrome: 167E

511.318804 : Scrubbing FRAD: 421B1C...
511.318868 : FAULT DETECTED! FRAD: 421B1C Word: 25 Bit(s): 30 31
511.318895 : WORD HAD MULTIPLE BITS UPSET WITH 2 ERRONEOUS BITS
511.318920 : Word: 25 | Expected: 0 Actual: C0000000
511.318949 : FAULT DETECTED! FRAD: 421B1C Word: 26 Bit(s): 0 1
511.318974 : WORD HAD MULTIPLE BITS UPSET WITH 2 ERRONEOUS BITS
511.318998 : Word: 26 | Expected: 0 Actual: 3
511.319088 : Scrubbing of FRAD: 421B1C Finished with 4 Upsets !

511.319133 : ACCIDENTAL CORRECTION:
        FRAD: 421B1C Word: 25 Bit: 30 Syndrome: 167E !

Clearing CRC latch !
After CRC Scrub, CRC = 1
```

```
************************************************************
Polling for 0 iterations, FIFO_COUNT: 4

511.389930 : CRC HIGH: Multi-Bit Upset! Syndrome: FF

511.390298 : Scrubbing FRAD: 421B1D...
511.390347 : FAULT DETECTED! FRAD: 421B1D Word: 25 Bit(s): 31
511.390371 : Word: 25 | Expected: 0 Actual: 80000000
511.390399 : FAULT DETECTED! FRAD: 421B1D Word: 26 Bit(s): 0
511.390425 : Word: 26 | Expected: 0 Actual: 1
511.390513 : Scrubbing of FRAD: 421B1D Finished with 2 Upsets !

Clearing CRC latch !
After CRC Scrub, CRC = 0
************************************************************
```

This selection presents a large upset spread across multiple frame. Each frame contains MBUs resulting from a single particle strike. The full readback scrub was necessary because all of the ECC information had been popped off of the FIFO, yet a CRC error still existed (also known as a CRC-Only error).

```
Frame_ECC 0: 80000001
Frame_ECC 1: 20001284
Frame_ECC 2: CRC = 1
Frame_ECC 3: 6
Frame_ECC 7: 12
Polling for 0 iterations, FIFO_COUNT: 6

211.494956 : CRC HIGH: Multi-Bit Upset! Syndrome: 1

211.495321 : Scrubbing FRAD: 1284...
211.495387 : FAULT DETECTED! FRAD: 1284 Word: 98 Bit(s): 6 7
211.495413 : WORD HAD MULTIPLE BITS UPSET WITH 2 ERRONEOUS BITS
211.495439 : Word: 98 | Expected: 0 Actual: C0
211.495524 : Scrubbing of FRAD: 1284 Finished with 2 Upsets !

Clearing CRC latch !
After CRC Scrub, CRC = 1
************************************************************
Polling for 0 iterations, FIFO_COUNT: 28

211.742929 : CRC HIGH: Multi-Bit Upset! Syndrome: 3
```

```
211.743138 : Scrubbing FRAD: 1285...
211.743192 : FAULT DETECTED! FRAD: 1285 Word: 98 Bit(s): 5 6
211.743217 : WORD HAD MULTIPLE BITS UPSET WITH 2 ERRONEOUS BITS
211.743241 : Word: 98 | Expected: 0 Actual: 60
211.743324 : Scrubbing of FRAD: 1285 Finished with 2 Upsets !

Clearing CRC latch !
After CRC Scrub, CRC = 1
**************************************************************
Polling for 0 iterations, FIFO_COUNT: 4

212.768017 : CRC HIGH: Multi-Bit Upset! Syndrome: C

212.768231 : Scrubbing FRAD: 1287...
212.768287 : FAULT DETECTED! FRAD: 1287 Word: 98 Bit(s): 5 6 7 8
212.768312 : WORD HAD MULTIPLE BITS UPSET WITH 4 ERRONEOUS BITS
212.768337 : Word: 98 | Expected: 0 Actual: 1E0
212.768420 : Scrubbing of FRAD: 1287 Finished with 4 Upsets !

Clearing CRC latch !
After CRC Scrub, CRC = 1
**************************************************************
Polling for 0 iterations, FIFO_COUNT: 0
212.909861 : Performing Full Readback...

**********Begin Frame Readback 0**********

213.203969 : Scrubbing FRAD: 1286...
213.204070 : FAULT DETECTED! FRAD: 1286 Word: 98 Bit(s): 6 7 8 9
213.204098 : WORD HAD MULTIPLE BITS UPSET WITH 4 ERRONEOUS BITS
213.204132 : Word: 98 | Expected: 0 Actual: 3C0
213.204226 : Scrubbing of FRAD: 1286 Finished with 4 Upsets !
213.204278 : 4 TOTAL UPSET(S) IN FRAME 1286

214.725186 : Full Readback 0 Finished with 4 total Upsets !

Clearing CRC latch !
214.725306 : Full Readback Finished !
**************************************************************
```

This final excerpt indicates a SBU followed by an even-numbered MBU. The scrubber detected the CRCERROR too soon and mistook the SBU to be an odd-numbered MBU. Hence, when it scrubbed the frame containing the SBU, there were no errors found as the

Readback CRC had already fixed the upset. The CRCERROR in this case came from the even-numbered MBU in a subsequent frame.

```
Frame_ECC 0: 930F1533
Frame_ECC 1: 60420F9C
Frame_ECC 2: CRC = 1
Frame_ECC 3: 3
Frame_ECC 7: 11
Polling for 0 iterations, FIFO_COUNT: 3

91.282858 : CRC HIGH: Multi-Bit Upset! Syndrome: 1533

91.283302 : Scrubbing FRAD: 420F9C...

!!!!!!!!!! 91.283414 : No errors found at FRAD: 420F9C !!!!!!!!!!

91.283455 : SBU corrected
        @ FRAD: 420F9C Word: 15 Bit: 19 CHECK NEARBY UPSETS!

Clearing CRC latch !
After CRC Scrub, CRC = 1
**************************************************************
Polling for 0 iterations, FIFO_COUNT: 4

91.354242 : CRC HIGH: Multi-Bit Upset! Syndrome: 3

91.354587 : Scrubbing FRAD: 420F9D...
91.354635 : FAULT DETECTED! FRAD: 420F9D Word: 15 Bit(s): 17 18
91.354658 : WORD HAD MULTIPLE BITS UPSET WITH 2 ERRONEOUS BITS
91.354716 : Word: 15 | Expected: 0 Actual: 60000
91.354813 : Scrubbing of FRAD: 420F9D Finished with 2 Upsets !

Clearing CRC latch !
After CRC Scrub, CRC = 0
**************************************************************
```