3-2022

Implementation and Characterization of AHR on a Xilinx FPGA

Andrew J. Dittrich

IMPLEMENTATION AND CHARACTERIZATION OF AHR ON A XILINX FPGA

THESIS

Andrew J. Dittrich, Captain, USSF

AFIT-ENG-MS-22-M-025

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-22-M-025

IMPLEMENTATION AND CHARACTERIZATION OF AHR ON A XILINX FPGA

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Andrew J. Dittrich, B.S.E.E.

Captain, USSF

March 24, 2022

AFIT-ENG-MS-22-M-025

IMPLEMENTATION AND CHARACTERIZATION OF AHR ON A XILINX FPGA

THESIS

Andrew J. Dittrich, B.S.E.E.
Captain, USSF

Committee Membership:

Maj. Nicolas Hamilton, Ph.D
Chair

Lt. Col James Dean, Ph.D
Member

Douglas Hodson, Ph.D
Member

Abstract

A new version of the Adaptive-Hybrid Redundancy (AHR) architecture was developed to be implemented and tested in hardware using Commercial-Off-The-Shelf (COTS) Field-Programmable Gate Arrays (FPGAs). The AHR architecture was developed to mitigate the effects that the Single Event Upset (SEU) and Single Event Transient (SET) radiation effects have on processors and was tested on a Microprocessor without Interlocked Pipeline Stages (MIPS) architecture. The AHR MIPS architecture was implemented in hardware using two Xilinx FPGAs. A Universal Asynchronous Receiver Transmitter (UART) based serial communication network was added to the AHR MIPS design to enable inter-board communication between the two FPGAs. The runtime performance of AHR MIPS was measured in hardware and compared against the runtime performance of standalone TMR and TSR MIPS architectures. The hardware implementation of AHR MIPS demonstrated flexible runtime performance that was nearly as fast as TMR MIPS, never as slow as TSR MIPS, and demonstrated performance in between those extremes. Hardware testing and verification of AHR MIPS showed that the AHR mitigation strategy presents a large performance tradespace, where a user can adjust both the runtime processor performance and radiation tolerance to fit the constraints of a space mission, while also continuing to provide adaptive performance based upon the current radiation environment.

AFIT-ENG-MS-22-M-025

For my family

Acknowledgements

I am thankful for being granted the opportunity to earn my Master's degree and further develop my technical abilities and knowledge. I am thankful for my family, because without their support this challenge would have seemed insurmountable. I would like to thank my research advisor for his guidance, patience, and support throughout my research.

Andrew J. Dittrich

Table of Contents

Page

Page

List of Figures

List of Tables

IMPLEMENTATION AND CHARACTERIZATION
OF AHR ON A XILINX FPGA

I. Introduction

1.1 Research Context

Electronic devices operating in space are subjected to a significantly more volatile radiation environment than terrestrial electronics. The Earth's atmosphere shields terrestrial electronics from high-energy cosmic particles, and the accumulation of radiation induced defects. Without protection from the atmosphere, electronics intended for use in space must be designed to comply with a different set of operational requirements than terrestrial electronics. Specifically, space electronics must be designed to be resistant or nearly immune to the harsh radiation environment of space. The design process used to make a radiation-tolerant or radiation-hardened (rad-hard) device varies in complexity depending on the device type. Simple, low-level devices composed of relatively few components can be re-designed to operate in space with less effort than complex devices such as processors.

A space system, such as a communication satellite, is composed of an electronic subsystem which must perform a specific set of tasks. If any component of the electronic subsystem is not designed to operate reliably in space, the operational capabilities of that space system are jeopardized. In the past, government space system missions were relatively simple and often only required the transmission of collected information to ground stations for data processing and decision making. However, modern government space systems have mission requirements that require advanced

electronics to enable on-orbit processing and decision making. Specifically, these modern space systems require advanced and state-of-the-art processing capabilities.

Any processor used in traditional government space systems must be designed to operate with near 100% reliability to support national security and defense objectives. Oftentimes, these processors must reliably operate for mission lifetimes greater than 5 years. Processors are complex systems composed of millions, if not billions, of lower-level electronic components. The component density and complexity of processors makes them especially susceptible to high energy cosmic particles, which can introduce errors into the device, and over time will alter the operational characteristics of the device until failure or cause instantaneous device failure [4, 47]. A short-term interaction between a high energy cosmic particle and a processor which causes an upset is referred to as a single-event-effect (SEE). The most common SEEs are single-event-upsets (SEUs) and single-event-transients (SETs), which are non-destructive interactions that change the state of a digital memory element [59]. The long-term effect of radiation on a processor is referred to as total ionizing dose (TID). TID degrades the electrical characteristics of the devices that make up a processor and eventually leads to device failure. A processor can reliably operate in space if it is designed/modified to tolerate or be completely immune to SEUs/SETs and have a high TID tolerance.

Development and manufacture of a new rad-hard processor requires millions of dollars and years of design time. Once a rad-hard processor is ready for use, that design's performance will lag by several processor generations when compared to current-generation commercial-off-the-shelf (COTS) high-performance processors. Despite the performance difference, there are many government space missions that require rad-hard processors and have the budget to acquire or procure one. However, due to evolving space system mission requirements, not all government space systems require

the near 100% reliability offered by an expensive rad-hard processor. Oftentimes field programmable gate arrays (FPGAs) can be used as a low-cost substitute in terms of cost and power consumption.

FPGAs are devices that can be programmed with application specific circuit designs and reprogrammed with a different design as the user sees fit. Since the operations performed by an FPGA device are specified by a user, the power consumed is much lower than if that operation were performed on a rad-hard processor, which may need to execute other processes just to keep the device operational. Oftentimes space systems do not require the generalized processing power of a rad-hard processor and can use an FPGA to efficiently perform only the operations necessary to support mission requirements. If a processor is required, soft-core processors can be implemented in the programmable fabric of an FPGA. Soft-core processors implemented on an FPGA are susceptible to the same radiation effects as normal processors. However, design methodologies that provide a normal processor radiation tolerance can also be employed for processors implemented on an FPGA. These design methodologies refer to methods of radiation mitigation and are the subject of this research.

To ensure reliable operation of an FPGA in space, the selected radiation mitigation method(s) must be well-proven and carefully integrated into the circuit design [37]. For this reason, research into the methods of radiation mitigation for FPGAs is a rich and evolving field. Radiation mitigation for FPGAs can be obtained using redundancy techniques or using radiation shielding. Radiation shielding requires that the FPGA be packaged using specialized protective materials, and this method is not ideal when space systems are highly constrained in terms of size and weight. This research only focuses on mitigation methods that use redundancy. Redundancy techniques are used to protect an FPGA against SEUs and SETs. Redundancy techniques cannot protect an FPGA against the effects of TID, because TID resistance is determined by the

process used to manufacture the device. Modern methods of radiation mitigation using redundancy can be split into three broad categories: hardware redundancy, software redundancy, and hybrid redundancy.

Hardware redundancy refers to methods that obtain redundancy through modification of the hardware design to detect and correct errors/faults. A commonly used method of hardware redundancy is Triple Modular Redundancy (TMR), which improves radiation tolerance by replicating a digital circuit three times and connecting all copies to a simple majority voter which determines the correct output [32]. TMR can be used to increase the radiation tolerance of an FPGA but does so at a cost. Specifically, TMR offers radiation tolerance at the cost of logical resources and power. If applied to an entire FPGA design, TMR would require over 3x the logic hardware resources to implement and would require more power to operate, than an equivalent non-TMR version of the design.

Software redundancy refers to methods that obtain redundancy by modifying the software that runs on a processor implemented in an FPGA. Temporal software redundancy (TSR) is a method that provides redundancy by performing each instruction a processor executes multiple times at different points during program runtime and comparing the results to determine the correct output. Like TMR, TSR can increase the radiation tolerance of a processor on an FPGA, but does so at the cost of processor performance. TSR increases the length of programs executed by a processor which requires more time to complete than an equivalent non-TSR program, and thus reduces processor runtime performance.

Hybrid redundancy methods are those that integrate multiple methods, such as TMR and TSR, into the same design. The use of TMR or TSR in an FPGA design to improve radiation tolerance is a common practice, however only one method is typically implemented. Selecting only one method limits the performance of a radia-

tion tolerant FPGA processor design, because both TMR and TSR provide different levels of radiation tolerance while providing different benefits. Hybrid methods seek to find an optimized design that can balance the benefits of both hardware and software redundancy, or any combination of other redundancy methods. However, hybrid methods are not common and any that do exist are highly application specific.

The decision to apply hardware, software, or hybrid redundancy to an FPGA processor design requires careful consideration of processing and power requirements. The chosen redundancy method must allow for reliable device operation under the expected worst-case radiation scenario that will be encountered on-orbit. However, the implemented redundancy method must also be power efficient since space systems operate under constrained power budgets. Once a redundancy method has been selected, it is applied to the FPGA during the design phase and does not change. By implementing a redundancy method which does not change, or is static, the performance characteristics of an FPGA-based processor do not change during runtime, even if the operational radiation environment changes for better or worse. This research is focused on further developing a hybrid redundancy method that is adaptive and enables a processor to have performance characteristics that change depending on the current radiation environment.

The Adaptive-Hybrid Redundancy (AHR) architecture was previously developed by Hamilton in [20] and can operate in two different modes, where each utilizes a different redundancy technique. The AHR architecture improves the radiation tolerance of a processor by using a controller to dynamically select between the two different operating modes, depending on the number of detected processor errors caused by radiation effects. Each operating mode implements a different radiation mitigation strategy: one using TMR and the other using TSR. The TMR operating mode offers the highest radiation tolerance and processor performance and is used

when the number of detected upsets is high. The TSR operating mode offers less radiation tolerance and processor performance, but consumes less power than the TMR operating mode. AHR specifically aims to improve tolerance to SEUs/SETs and does not improve a processors TID tolerance. AHR stands as an important development because, based upon a literature review, no other hybrid and adaptive radiation mitigation strategy for processors has been physically implemented on an FPGA [27]. This research seeks to further develop AHR and verify the operation of AHR when implemented on a COTS FPGA.

One of the goals defined in the 2020 Defense Space Strategy created by the Department of Defense (DoD), was that the DoD will, "develop an agile space enterprise that can take advantage of emerging technological and commercial innovation in order to continually outpace adversary threats" [56]. The AHR architecture was designed to be generic so that it could be integrated into the design of a processor implemented on any FPGA device, even as FPGA technology continues to advance. The core design philosophy of AHR supports DoD goals to leverage lower-cost COTS electronics, which can meet the processing performance needs of current and next-gen space systems, while also decreasing the time required to field such systems. Experimental verification of AHR, as applied to a processor on an FPGA, would determine the mitigation strategy's feasibility as a low-cost alternative to rad-hard processors and would mature the technology for integration into future space systems.

1.2 Research Questions

This research seeks to answer the following questions:

(a) Can AHR be implemented on a COTS FPGA(s)?

(b) What are the performance characteristics of AHR on a COTS FPGA? (Processor Performance)

(c) How does the measured performance of AHR compare to predictions and results produced in prior research?

1.3 Research Assumptions

This research is a continuation/extension on the work in [20]. Therefore, this research makes many of the same assumption while also including additional assumptions to maintain focus and scope.

1. The only radiation effects considered are SEUs and SETs [20].
   (a) No other types of radiation effects occur.
   (b) Multiple-bit upsets (MBUs) where a single radiation strike causes errors in multiple adjacent registers are so unlikely that they do not occur.
2. The processor refers only to the Controller and Datapath of a processor [20].
   (a) The Controller is composed of a finite state machine (FSM) which translates instructions into control signals for the Datapath.
   (b) The Datapath consists of general purpose registers (GPRs), a program counter (PC) register, logic to update the GPRs and PC register, an ALU to process data, and logic to control data flow. The operation of the Datapath is controlled by control signals from the controller.
3. The processor and all lower-level components contained are subject to SEUs and SETs with the following exceptions [20].
   (a) The TMR Voter in AHR MIPS described in Ch. II Section 2.6.1.2 is immune to errors.

(b) The AHR Controller in AHR MIPS described in Ch. II Section 2.6.1.3 and the multiplexers used by the AHR Controller for signal routing are immune to errors.

4. Memory refers to the location where instructions and data are stored for use by the processor. The processor accesses memory through read and write operations [20].

5. Memory is immune to all errors caused by radiation effects [20].

   (a) Memory hardening is not within the scope of this research

   (b) Memory may be hardened by error correcting codes (ECCs), redundancy, shielding, or any combination of these methods.

6. The codes used in communication operations between the Processor and Memory are immune to all errors caused by radiation effects.

   (a) These codes are short-constant values which can be made immune to errors using ECCs.

1.4 Document Overview

The research presented in this document is ordered as follows:

(a) Chapter II presents the background research performed to understand this topic.

(b) Chapter III presents the methodology employed to execute the proposed AHR R&D.

(c) Chapter IV presents and discusses the results of the AHR R&D.

(d) Chapter V presents conclusions, contributions of this research, and discusses future work.

II. Background

2.1 Introduction

This chapter presents the background necessary to understand the design/development of AHR, and why it is important to further verify AHR in FPGA hardware. Section 2.2 provides a high-level explanation on the different categories of radiation effects electronics experience while operating in space. Section 2.3 discusses the different types of modern FPGA technologies, highlights why SRAM FPGAs are commonly used in space, and further explains the radiation effects to which they are susceptible. Section 2.4 summarizes the variety of methods used to mitigate the radiation effects experienced by SRAM FPGAs. Section 2.5 describes the physical and simulated radiation testing methods employed to verify and validate the functionality of a given mitigation technique. This section provides insight into the testing method selected for this research. Section 2.6 provides an in-depth discussion on AHR which highlights why it is a novel technique to detect and correct radiation induced upsets and why it is the foundation upon which this research is built.

2.2 Radiation Effects on Electronics in Space

Electronic devices used in space are subjected to an operational environment vastly different than what is experienced terrestrially. The lack of an atmosphere in space requires different device manufacturing and assembly procedures to ensure the device can even operate. The lack of atmosphere also exposes electronic devices to high-levels of radiation, which can damage the device or degrade device performance by upsetting the device's operation during runtime. The space-radiation environment can generate short-term and long-term effects that affect device performance.

2.2.1 Long-Term Effects

The most prevalent long-term radiation effect on electronic devices is Total Ionizing Dose (TID) [59]. TID accumulates in silicon devices operating in a radiation environment, and degrades the electrical parameters of that device until it no longer functions as expected or at all [1, 28, 58]. The TID tolerance of a device is determined by the manufacturing and assembly process. For this reason, TID mitigation is implemented through physical means, such as the addition of shielding to reduce the amount of radiation the device encounters. Although shielding can be used to mitigate TID, it is costly to implement in space systems where the size and weight of every included component is highly constrained. Electronic devices manufactured to have a higher TID resistance are usually preferred for use over shielding, but are specialized products that come with a higher unit cost.

2.2.2 Short-Term Effects

Short-term radiation effects are classified as a type of Single-Event Effect (SEE). SEEs include radiation strikes that can lead to destructive and non-destructive events. Destructive SEEs include Single-Event Latchup (SEL), Single-Event Gate Rupture (SEGR), and Single-Event Burnout (SEB) [28, 59]. These effects are caused by a radiation strike on the electronic substrate of a device, which creates an unintended low-impedance/high-current path, that can lead to damage or destruction of the device. Non-destructive SEEs include Single-Event Upsets (SEUs) and Single-Event Transients (SETs) [1, 28]. Both SEUs and SETs are caused when a radiation strike generates an electronic charge large enough to change the logic state of a low-level electronic component within an FPGA device (transistor, flip-flop, register) [4, 47].

2.3 FPGAs In Space

FPGAs are commonly used in space systems due to their low-design cost when compared to the purchase of a radiation-hardened processor or cost to develop and manufacture an Application-Specific Integrated Circuit (ASIC). There are three main types of modern FPGAs: antifuse, flash memory-based, and SRAM-based. Selecting the type of FPGA to use in a space system depends on a variety of factors including mission lifetime, required computational performance, power usage, and cost. The difference between each type of FPGA is the method in which the configuration of the device is maintained. Antifuse FPGAs store device configuration in fuse-based memory cells, which are one-time programmable [1, 47]. Antifuse FPGAs are less susceptible to radiation effects because the configuration of the device is physically fixed in hardware after being programmed once. Flash memory-based FPGAs store configuration memory in non-volatile flash memory cells and can be reconfigured using a bitstream. These kinds of FPGAs have been found to be unreliable for space missions with a long lifetime, due to, "low immunity to TID and SELs" [47]. SRAM-based FPGAs are similar to Flash-based FPGAs, but instead store configuration data in volatile SRAM memory cells. SRAM FPGAs are desirable for use in space because they offer high performance and ease of reconfigurability. However, SRAM-based FPGAs are more susceptible to radiation induced upsets than the other FPGA types. Despite being more susceptible to radiation induced upsets, SRAM-based FPGA performance is so high, that it is worth the effort and cost to design strategies/methodologies that can increase the radiation tolerance of these devices. This research is entirely focused on further developing a mitigation strategy that enables a high-performance SRAM-based FPGA to operate in a radiation environment.

Section 2.3.1 provides details on why SRAM FPGAs are desirable for use in space systems and why they are susceptible to radiation effects. Sections 2.3.2 - 2.3.5

describe the specific radiation effects that must be considered or dealt with in order for an SRAM FPGA to function reliably in space.

2.3.1 SRAM FPGAs

SRAM FPGA performance is high because these devices are manufactured using standard CMOS processes [58]. CMOS fabrication is an advanced process, which can create FPGA devices that are transistor dense, thus offering higher computational performance than a flash or antifuse FPGA device. SRAM FPGAs are easily reconfigurable because each SRAM cell in the device can be individually programmed and reprogrammed using a bitstream generated by any of the industry standard FPGA design suites. SRAM FPGAs afford mission planners and hardware designers the flexibility to perform missions beyond what the device was originally programmed to do [1]. However, SRAM FPGAs are more susceptible to short-term radiation effects than the other FPGA types because the state of an SRAM cell is volatile and can easily be changed due to a radiation strike.

2.3.2 TID

SRAM-based FPGAs are susceptible to the effects of radiation-induced charge trapping within the oxide layers of the CMOS devices that constitute the FPGA [58]. Trapped charge in the oxide layers of CMOS devices causes ionization to occur, which causes performance degradation due to increasing leakage currents and other effects [47]. The accumulation of TID over time will cause SRAM-based FPGAs to fail. As mentioned in Section 2.2.1, the TID tolerance of an SRAM-based FPGA is determined by the manufacturer. The Xilinx Virtex-5QV FPGA device is a modern example of an FPGA that is manufactured to be Rad-Hard By Design (RHBD) and thus has a high TID tolerance [60]. The Virtex-5QV FPGA is manufactured using

a 65-nm technology process, which is many generations behind current generation high-performance COTS devices, such as the Virtex Ultrascale+ built using a 16-nm process [62]. Not only is the performance lower, but RHBD SRAM FPGAs are much more expensive to produce due to the specialized process required to make RHBD CMOS devices. This means that RHBD SRAM FPGAs present a much lower cost-performance ratio than most COTS FPGAs. However, RHBD FPGAs such as the Virtex-5QV are the only option when an FPGA must perform with near immunity to all radiation effects. This research does not seek to mitigate the buildup of TID in SRAM FPGAs, but instead focuses on mitigating short-term radiation effects, which are more cost-effective to mitigate than TID.

2.3.3 SEUs & SETs

SEUs and SETs are the most common SEEs that affect SRAM-based FPGAs. SEUs are single particle events that change the value held in a digital memory element, and SETs are radiation strikes that modify the behavior of a logic cell [19, 28, 59]. These radiation effects are common in SRAM FPGAs because these devices are densely packed with bi-stable volatile memory and logic elements. Despite being the most common SEEs, SEUs and SETs are considered soft errors that can be cleared/dealt with as the event occurs during device runtime [4]. These soft errors, also called transient errors, are non-permanent device faults that are highly localized and can be corrected without a system reset.

SEUs can have various effects on the operation of a circuit on an FPGA depending on the memory element that is struck by a particle. The effect of lowest consequence occurs if a memory element that stores user/runtime data is struck. In this case, the output of any operations using that memory element will be incorrect. This error can be simply corrected by overwriting the erroneous data with corrected values. The

most substantial consequence occurs if a memory element that stores FPGA circuit configuration data is struck. In this case, the operation of the circuit is fundamentally changed, and can produce unexpected behavior. This error must be corrected by re-loading that configuration memory cell with an error-free version. This is known as configuration scrubbing and is discussed in Section 2.4.1.4. In either case, the memory element that experienced an SEU is not permanently damaged and can be expected to operate correctly after the error is corrected.

Since SETs are defined as only affecting FPGA logic cells, they cannot change the fundamental configuration of the FPGA. SETs introduce highly transient errors to the combinational logic used by the circuit and can result in an error at the output of the affected logic. Within a chain of combinational logic there is some inherent protection against SETs creating an output error. The effects of a SET can be masked, or ignored, either electrically, logically, or temporally. A SET is electrically masked when the radiation strike produces a logic pulse of 1 within combinational logic, but does not persist through enough successive components/gates to manifest a final output error. A SET can be logically masked if the produced logic pulse of 1 encounters a logic gate whose output, at that moment, cannot change due to a single received logic 1 value. For example, if a SET produced logic pulse of 1 is an input to an AND gate and the other input is a logic 0, then the SET does not continue to propagate. Temporal masking occurs when the SET generates a logic pulse of 1 and is received by a component that ignores the errant input, because that component only reads input values under specific conditions. For example, a SET produced logic pulse of 1 would be ignored by a D flip-flop if the pulse arrived at the input at any point other than the rising edge of the clock. If a SET-produced error is not masked, then the resulting output of the affected combinational logic will have an error that could affect future operations. In the same manner as SEUs, an FPGA logic element

that experiences a SET is not permanently damaged and will continue to function properly after the error is corrected [7, 20, 33].

These soft errors typically only affect a single memory/logic element and thus only generate a Single-Bit Upset (SBU). However, if the radiation strike has sufficient energy, a Multi-Bit Upset (MBU) can be generated. Therefore, all methods of error detection and radiation mitigation must be designed to also account for MBUs, despite MBUs accounting for a small fraction of observed total SEU rates in space systems [4]. This research is focused on the detection and mitigation of SEUs and SETs, and thus a redundancy scheme was created that provides protection against SBUs. The redundancy scheme used in this research combines two redundancy methods. Each method is implemented in unique ways and as a result, each method also provides varying degrees of protection against MBUs. The first method has higher inherent protection against MBUs because it is a technique that provides redundancy by physically separating circuit components in hardware. If there is more physical separation between components in a circuit design, then a MBU is less likely to affect more than one logic cell or group of cells that define a register for example. The second method has lower inherent protection against MBUs because it provides redundancy through time by repeating operations and comparing the results of those operations. The second method provides less physical separation between circuit components, and as a result could be more susceptible to a MBU manifesting errors in more than one logic cell or group of cells.

2.3.4 SEFI

A Single-Event-Functional Interrupt (SEFI) is another type of soft-error, where a radiation strike changes the logical state of a critical memory element or register which interrupts the normal operation of the device [29]. In an FPGA, a radiation

strike to a system control register may cause an SEFI to occur [4]. A radiation strike to a register that defines a program branch target can lead to an illegal branch being taken and is also considered a SEFI. An uncorrected SEFI may persist until the device is reset, because the affected logical element may not be accessed as often as elements used to define user memory [29]. Although SEFIs have a profound impact on device operation, they are still a soft-error, and do not permanently damage the affected elements or device. This research implements a redundancy/mitigation scheme that operates in two modes. Only one of the operational modes can detect errors that affect branch instructions or could cause an illegal branch to occur. The other operational mode does not provide any protection against branch instruction errors, and thus the overall redundancy architecture only provides partial protection against SEFIs.

2.3.5 SEL, SEGR, and SEB

SRAM-based FPGAs are susceptible to potentially destructive Single-Event-Latchup effects (SELs). A SEL is a semi-permanent error, that can destroy the element(s) that experience the radiation event. SEL is caused by an energetic particle interaction within a CMOS device. This interaction can create a path for current within the parasitic structure present in CMOS devices as a result of having NMOS and PMOS transistors next to each other [10, 20]. This parasitic current path is low-impedance and can result in device burn-out if it is not power-cycled. SELs can be detected and corrected using current sensors and other monitoring devices included as part of the FPGA or device packaging. Hardening solutions for SELs are dependent on the manufacturing process used for the FPGA. Methods used to harden against SEL include: use of epitaxial layers, guard rings, trench isolation, and Silicon-On-Insulator (SOI) technologies, all of which are discussed in depth in a study by Brugier et al. in [10].

Single Event Gate Rupture (SEGR) and Single Event Burnout (SEB) are permanent errors that are caused when a radiation strike has high enough energy to immediately destroy elements/devices within an SRAM-based FPGA. SEGR is caused when a particle strike causes the insulating layer between the gate and device dielectric to no longer function, thus causing device failure [20]. SEB occurs when a radiation strike creates an un-correctable and unintended current path within the parasitic structures inherent to CMOS devices and causes device burnout/destruction. [20, 55].

The redundancy methods used in this research do NOT protect an SRAM-based FPGA against SEL, SEGR, or SEB.

2.4 Methods of Mitigation for SRAM FPGAs

SRAM-based FPGAs have been heavily researched and radiation tested both terrestrially and on-orbit because these devices offer high performance and ease of reconfigurability [11]. Many SRAM-based FPGA devices have in-depth and validated radiation performance characteristics [1, 28, 57]. However, it has been experimentally shown that SRAM-based FPGA performance in space is greatly improved by using radiation mitigation techniques. As a result, there is great research emphasis on creating new and novel design methodologies to mitigate and correct errors caused by radiation effects for SRAM FPGAs. Methods of radiation mitigation and correction for SRAM FPGAs can be split into three broad categories: Hardware methods, Software methods, and Hybrid methods. Hardware methods usually provide higher mitigation capabilities and consume more power, as compared against software methods where less effective mitigation is provided while consuming less power. Hybrid methods combine two or more methods to optimize the balance between mitigation capabilities and power consumption. Within each category there exist a variety of different mitigation methodologies, which have reached a high-level of development,

with some having been proven on-orbit as part of space-systems.

Section 2.4.1 discusses hardware methods that modify the physical design of the circuit to increase radiation tolerance. Section 2.4.2 discusses software methods that modify the programs or software architecture running on the device to increase radiation tolerance. Section 2.4.3 discusses hybrid methods that combine hardware and software methods to increase radiation tolerance. This research further develops a unique method of hybrid redundancy called Adaptive Hybrid Redundancy (AHR). Section 2.6 will provide an in-depth discussion on the development of the novel AHR architecture, the ways in which the operation of AHR was verified, and how the AHR architecture compared against other radiation mitigation methods.

2.4.1 Hardware

Hardware redundancy refers to any redundancy method that requires modifications be made to the hardware design of the circuit to detect radiation induced errors during runtime. Hardware redundancy is typically implemented through spatial redundancy combined with configuration scrubbing [47]. However, hardware redundancy can also be implemented using Error Correcting Codes (ECCs).

Spatial redundancy increases radiation tolerance by duplicating either specific modules of the circuit design or the entire circuit multiple times in the programmable fabric of an FPGA. The duplicated circuit elements are typically connected to a voter circuit, which is used to detect any errors that occur, and initiate the appropriate process to correct that error. Methods of spatial redundancy are described in Sections 2.4.1.1 & 2.4.1.2 and more advanced techniques are described in Section 2.4.1.3.

Configuration scrubbing refers to the process used to correct any errors that affect the configuration memory of an FPGA device. Section 2.4.1.4 discusses the different methods and benefits of scrubbing. An FPGA circuit design that uses some form

of spatial redundancy specifically paired with scrubbing decreases the likelihood that a radiation induced upset (SEU or SET) generates a disruptive error during circuit operation.

ECCs, also called Error Detection and Correction (EDAC) codes, are used to protect data against errors/upsets that occur in memory or in transmission channels [45]. Various types of ECCs are discussed in Section 2.4.1.5.

2.4.1.1 TMR

The most common duplication scheme for hardware redundancy is Triple Modular Redundancy (TMR) [32, 47]. TMR is a static redundancy scheme where the specified circuit modules are triplicated in the FPGA programmable fabric. TMR is a static redundancy scheme because the device's radiation tolerance does not change during operation after implementing the triplicated modules in the FPGA fabric [32]. The TMR scheme has each triplicated module connected to either one majority voter, or to a network of triplicated voters to further increase circuit reliability [32]. By connecting modules to a voter, single faults/errors caused by a radiation upset are effectively masked. The voter masks errors by constantly monitoring the output of each module, and if an error is found in one of the modules, the voter can use the output of the other two modules in agreement, while ignoring the module in error. A diagram of a TMR Microprocessor Without Interlocked Pipeline Stages (MIPS) architecture is shown in Figure 1 as an example of the redundancy scheme.

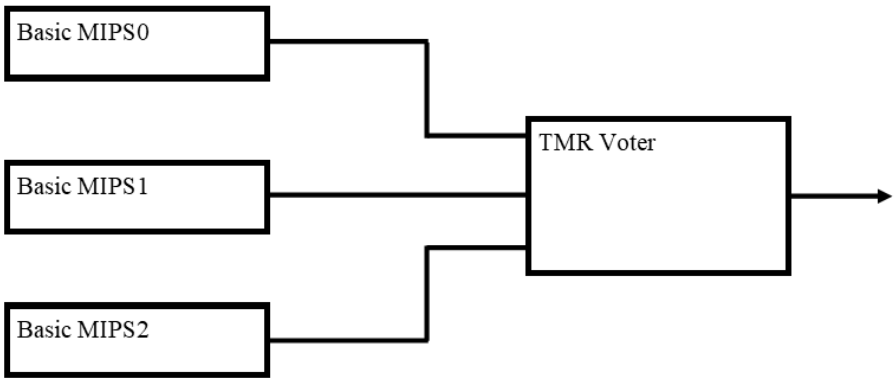

Figure 1. TMR MIPS Simplified Block Diagram

TMR is a popular choice for improving the redundancy of a circuit design not only because of the error masking capabilities, but also because of the relative ease of usability in any circuit design. Implementing TMR in a circuit design only requires module duplication, and insertion of voter circuitry. These design modifications are relatively easy to implement, with several commercial and academic software tools available for automating this process [47]. Although TMR is not complex to implement it does come with cost tradeoffs.

First, TMR is costly in terms of FPGA logical fabric resources. Any given module to which TMR is applied will be triplicated, and therefore requires at a minimum 3x the FPGA resources to implement [32]. Voter circuitry must also be inserted which requires more FPGA resources. By consuming more FPGA resources, TMR also increases the amount of power used by the FPGA device. The high cost of TMR must be considered by mission planners/designers when FPGA resources and the power budget of a space system are constrained.

Despite the high cost of implementation, the reliability improvements gained by

using TMR are so great that implementation of the technique at any level is beneficial. In a study performed by [32], the reliability improvements vs. resource utilization of TMR was compared against other well-known techniques in custom circuit technologies: quadded logic, state machine encoding, and temporal redundancy. The results of the study were that none of the other techniques could provide the same level of reliability as TMR and often incurred higher resource utilization than TMR [32]. Since TMR is one of the best methods for providing hardware redundancy against radiation induced upsets/errors, it is often important to at least triplicate critical modules in a circuit design. Selective TMR is a method used to determine what the critical modules are in a design and only apply TMR to those modules. Selective TMR is discussed further in Section 2.4.1.3.

TMR is an effective method for increasing the reliability of a circuit design implemented on an FPGA, intended for use in a radiation environment. A standalone TMR architecture can detect both single-bit and multi-bit errors and can correct single-bit errors. TMR does incur a high resource and power cost, however if it cannot be fully applied to a circuit design, it is still important to apply TMR selectively on critical circuit components. The redundancy architecture developed in this research implements full TMR, specifically for a MIPS processor as will be discussed in Section 2.6.

2.4.1.2 DMR

Although TMR is a powerful means for implementing radiation mitigation, the associated power and hardware resource cost may be too great for certain applications. In these cases, Dual Modular Redundancy (DMR) is an option which provides some radiation mitigation. DMR uses two redundant circuit modules connected to a voter to detect errors, and requires up to 33% less hardware resources than TMR [53].

A simple implementation of DMR is only able to detect errors and cannot mask errors as was described for TMR. This is because if an error is detected in one of the two redundant copies, then the system must either reset to a previous state or re-do the operation that used the errant module. This method alone provides some radiation mitigation, but would negatively affect the performance of an FPGA device in a high-radiation environment. However, there have been DMR architectures created that have increased mitigation capabilities by implementing additional logic on the output of either the DMR modules or DMR voter [2, 38, 63].

The redundancy architecture developed for this research does not use DMR, but DMR could be implemented into the architecture to improve performance during specific scenarios where a radiation strike has produced a permanent error in one of the TMR modules. This is described in Ch. V Section 5.2

2.4.1.3 Optimizing Spatial Redundancy

The biggest drawback of applying full TMR or DMR to an FPGA design is the amount of hardware resources consumed. However, the resources consumed can be reduced by evaluating which components within a given FPGA design actually require spatial redundancy. P. Samudrala et al. in [43] presented the Selective Triple Modular Redundancy (STMR) design technique, which applies TMR only to gates that have been determined to be sensitive to SEUs, using a specialized algorithm. Pratt in [39] also developed both an approach and software tool to decompose a circuit design into structures that are classified based upon their importance within the design, and selectively apply TMR to the most important structures until FPGA resource utilization is maximized. Another method of selective hardware redundancy was shown by S. Bahramnejad et al. in [3]. This method determined the System Failure Rate (SFR) of a given FPGA design, and iteratively applied hardware redundancy

to components in the design to reduce the SFR [3]. All these methods are considered static optimizations, meaning that the final design is determined after analysis and cannot be easily changed during runtime.

Another means of optimizing spatial redundancy is through adaptive redundancy. These are techniques that are dynamic and can change the level of hardware redundancy for the FPGA design based upon detected radiation upset rates. Most of these techniques are reliant on FPGAs that support partial reconfiguration (PR). An FPGA that supports PR allows a user to modify portions of an FPGA's configuration while the remainder of the FPGA continues to operate. This capability was a key feature in the adaptive architectures described in [16, 17, 26]. These studies used a radiation sensor connected to a specialized controller to either increase or decrease the number of redundant hardware modules, with respect to the detected number of radiation induced errors, using FPGA partial reconfiguration.

The performance flexibility gained by using a redundancy architecture on an FPGA that can change depending on the current radiation environment was a fundamental driver in the development of AHR, discussed in Section 2.6.

2.4.1.4 Scrubbing

The spatial redundancy methods previously described improve the radiation tolerance of an FPGA design by observing the output of logical operations in the FPGA to determine if an error occurred and perform error correction. However, these methods are unable to correct errors that accumulate in the configuration memory of the FPGA. The configuration memory defines the operation of the circuit design implemented on the FPGA. A radiation induced upset in configuration memory can modify the operation of the circuit and can cause an incorrect output to be generated until the configuration memory is corrected. Although spatial redundancy methods such

as TMR can mask an error produced by a single errant module, they cannot correct an error in the configuration memory of an FPGA [32]. One method for correcting configuration memory is called configuration scrubbing. Configuration scrubbing uses an, assumed to be, error-free copy of the original configuration FPGA bitstream to periodically re-write configuration memory on the device [39]. The error-free copy of the original FPGA configuration is typically stored off-chip in radiation hardened memory and is commonly referred to as the "golden copy." Periodic configuration scrubbing is essential for preventing the build-up of radiation induced errors/faults in the FPGA, and reducing the time an invalid circuit is allowed to operate [1].

FPGA configuration memory scrubbing can be implemented using either an external or internal scrubber. An external scrubber uses a highly radiation tolerant off-chip resource to perform the scrubbing operations. An internal scrubber instantiates the circuit required to perform scrubbing operations on the FPGA fabric. Whether the scrubber is external or internal, both methods use a scrubbing circuit to perform a readback of device configuration memory. The configuration read from memory corresponds to a specific portion of the FPGA design. The current configuration of that portion of the FPGA is compared against the corresponding configuration data read. If an error is detected during this comparison, the configuration memory at that location of the FPGA is corrected using the "golden copy" [50].

Both scrubber implementations perform the same operation, but each have different operational characteristics. An external scrubber uses no FPGA resources, but uses more physical space, is more radiation tolerant, and performs scrubbing operations slower than an internal scrubber. An internal scrubber has the scrubbing circuitry closer to the FPGA configuration memory and therefore performs scrubbing faster [50]. However, the internal scrubber circuitry itself is susceptible to errors in configuration memory, and is therefore unable to detect certain errors that could

modify its own operation, and thus is less radiation tolerant [50]. A study by Berg et al. experimentally found that an external configuration scrubber performed better than an internal one under the same radiation conditions for a Xilinx Virtex-4 FPGA [6].

By combining a method of spatial redundancy and configuration scrubbing, the reliability of an SRAM-based FPGA is drastically improved. For this reason, configuration scrubbing paired with TMR is commonly implemented on FPGAs to provide runtime data and configuration memory fault tolerance. This research recognizes the importance of configuration scrubbing, but does not currently implement it. This is because AHR needs to demonstrate correct operation in hardware before adding further capabilities, such as configuration scrubbing.

2.4.1.5 ECCs

Error Correcting Codes (ECCs) are typically implemented by modifying the data stored/transmitted in a system to include extra bits called check bits, which also requires that changes be made to the communication buses transmitting that data. As a high-level summary on the usage of ECCs, any data that needs to be sent is first encoded to produce a specified number of check bits. These encoded check bits are attached to the original data and transmitted to a receiver. The receiver will then perform its own encoding of the data bits received and determine if the re-produced check bits match those that were sent with the data. Systematic codes refer to any ECCs that maintain separation between the data bits and check bits as described in the generalized example given earlier [45]. Non-systematic codes are ECCs where the data bits themselves are fully encoded to produce a unique data packet that must be entirely decoded [45, 48].

One of the most prevalent ECCs used are Hamming codes. Depending on the com-

plexity of the Hamming code scheme implemented, Single Error Correction (SEC) up-to SEC and Double Error Detection (SEC-DED) can be performed [24]. This means that in the context of a radiation environment, Hamming codes can only be used to correct for a SBU. If a MBU affects multiple adjacent bits within a data word, then there are cases where Hamming codes will not detect those errors. However, a solution that can perform Double Adjacent Error Detection (SEC-DAED) and even Triple Adjacent Error Detection (SEC-TAED) was presented in [44]. The decoders and encoders used to implement Hamming codes can either be in hardware or software. A comparison between a hardware and software implementation performed in [46] found that, although a hardware implementation has superior reliability, a software implementation can be used as a low-cost solution to improve system reliability.

A summary of other modern ECCs is given in [48], however only Hamming codes are relevant to this research. The redundant architecture implemented as part of this research, assumes the use of Hamming codes within the communication network to satisfy the last research assumption given in Ch. 1 Section 1.3.

2.4.2 Software

Software redundancy refers to any redundancy method that requires modifications be made to the programs/software that operate on a given FPGA circuit. A common approach to software redundancy is through the use of duplicated instructions, as described in Section 2.4.2.1. Another method of software redundancy is to use signature analysis to determine if a block of code has been executed properly. Section 2.4.2.2 discusses signature analysis techniques. This research implements a new hybrid redundancy scheme in hardware that uses both TMR and the method of software redundancy described in Section 2.4.2.1.

2.4.2.1 EDDI

Software redundancy can be obtained through implementation of methods that provide redundancy in time, called Temporal Software Redundancy (TSR). The concept of TSR is, "to repeat computations in such a way that errors can be detected by comparing computation results" [36]. TSR methods have been shown to provide a high-level of fault detection, while only requiring code modifications and no hardware modifications [12]. For this reason, TSR methods are typically used when a design is resource constrained on an FPGA and cannot implement a method of spatial redundancy. However, as compared to spatial redundancy methods, TSR methods suffer from additional execution time overhead and reduce processor performance, because duplicate instructions are either added or repeated. The research presented in this document uses the concept of Error Detection by Duplicated Instructions (EDDI) to provide TSR [36].

EDDI is a software redundancy method that duplicates each instruction in a program during compilation, while using separate registers and variables for each instruction in a pair [36]. By using separate registers and variables, each instruction is not only executed at a different point in time, but the results of each operation are stored in physically separate locations in register space. This method provides redundancy because a radiation strike will typically only affect a small area on a device. Thus, the generated bit-flip will likely only modify the data stored in a single register used by one of the duplicated instructions in a pair. An EDDI instruction set will detect this bit-flip using an added comparison instruction, specifically a Branch if Not Equal (BNE), before writing any user registers to memory. The comparison instruction is used to detect whether an error has occurred and affected one of the registers in a pair. Error detection is performed only before writing a register value to permanent/main memory, because any error produced by an intermediate computation will propagate

to the final result that is written to memory [36].

If an error is detected by the BNE instruction comparing the data held in user registers, the BNE instruction branches to error recovery code. The method of error recovery used in EDDI is determined by the user, and the constraints of the system EDDI is implemented on. Since this research uses EDDI TSR, the method of error recovery selected was to reload/revert all user registers and the program to an earlier state stored as a "Save and Restore Point" (SRP). Further details on this process are given in Section 2.6.3.2. An example of EDDI TSR applied to a single instruction (Table 1) is shown in Table 2.

Table 1. Basic MIPS ADD Instruction [36]

ADD R3, R1, R2

Table 2. EDDI TSR MIPS ADD Instruction [36]

ADD R3, R1, R2	Main Instr.
ADD R23, R21, R22	Secondary Instr.
BNE R3, R23, gotoErrorRec	Comparison Instr.

EDDI TSR is a software redundancy scheme that is effective and relatively simple to implement at a high-level. However, the performance cost overhead associated with processing an EDDI TSR program is significant. For this reason an improved scheme was created by G. Reis et al. called Software Implemented Fault Tolerance (SWIFT) [40].

SWIFT uses duplicated instructions to provide redundancy, but has greater performance than EDDI TSR by reducing memory usage. EDDI duplicates all instructions

including store word (SW) instructions, and thus creates redundant copies of data in memory as well. SWIFT eliminates all duplicate SW copies by assuming that the memory connected to the processor already incorporates ECCs or some other form of redundancy [40]. A SWIFT program contains 50% less SWs than an equivalent EDDI program, and thus improves processor performance by reducing the number of interactions with main memory. SWIFT also incorporates a software only signature-based control-flow checking scheme to further improve the fault coverage of an EDDI-like redundancy scheme [40].

As previously mentioned, this research uses EDDI TSR, but specifically uses an EDDI-SWIFT hybrid TSR scheme. Like SWIFT, there are no duplicate SW instructions; but unlike SWFIT, there is no signature-based control-flow checking. This research assumes that all memory modules are radiation hardened, and because of this, the performance benefits of SWIFT over a standard EDDI TSR implementation were obtained [40].

2.4.2.2 Signature Analysis

Software implementations of signature analysis techniques can provide software redundancy by comparing pre-compiled signatures against signatures computed during runtime [34]. N. Oh et al. developed the first purely software implementation of signature analysis called Control-Flow Checking by Software Signatures (CFCSS) in [35]. That research defined the basic flow of signature analysis techniques to be [18, 35]:

1. Decompose a program into basic functional blocks (Code w/o a branch instr.)
2. Generate a reference signature associated with each block during pre-compilation
3. Store reference signatures in memory

4. Re-compute signatures of functional blocks during runtime

5. Compare corresponding signatures to detect an error

The basic signature analysis scheme defined above, can detect radiation induced errors that occur within a basic block, but cannot detect errors that affect branch or jump instructions, since they are not included as part of the signatures that are created. In research performed by B. Nicolescu et al. a more robust signature analysis technique was developed and tested to account for radiation effects that generate errors in branch instruction targets [34]. Another approach used to improve the fault detection of basic signature analysis was presented by O. Goloubeva et al. in [18], which added assertions to the basic blocks that could detect faults affecting branches.

Signature analysis techniques provide physically constrained systems with a means to implement some form of radiation mitigation. However, as was seen with EDDI TSR, these methods incur a code size overhead and thus reduce system performance. The reduction in performance using signature analysis scales with the number of branch instructions in the program [35]. This research does not utilize signature analysis techniques, because the preferred method of software redundancy was previously selected to be EDDI TSR [20].

2.4.3 Hybrid Redundancy

Hybrid redundancy methods refer to methods that implement a combination of hardware and software redundancy. Hybrid methods are implemented to leverage the benefits of both hardware and software redundancy. Hardware redundancy provides increased resilience to radiation effects at a higher resource utilization and power cost. Software redundancy lowers processor performance, operates at lower power, and offers less resilience as compared to hardware methods. For SRAM-FPGAs intended for use in space, there exists a large application domain where radiation resilience must

be considered along with system cost, power, and performance [14]. The large trade space that mission planners need to consider often leads towards optimized mitigation strategies that implement some form of hybrid redundancy [14]. Oftentimes these developed hybrid redundancy solutions are non-generalizable, because of the diversity of options for combining hardware and software redundancy methods.

A hybrid redundancy method was developed in [8], that combined the use of a specialized hardware watchdog monitor and software method that combined EDDI and control flow checking. The research in [8] showed promising results for the legitimacy of hybrid redundancy methods applied to a processor as part of a System-on-Chip (SoC). In that research high fault detection capabilities were provided with reduced software and memory overhead as compared to pure software approaches. However, implementation of the described hybrid redundancy approach required extensive changes to the programs that ran on the processor, and required integration of specialized infrastructure IP cores [8]. The methodology was not experimentally tested on an SRAM FPGA, but still provides valuable insight on the performance benefits gained using a method of hybrid redundancy.

The research presented in [14] developed a hybrid redundancy architecture that can be applied to ASICs and FPGAs. The architecture that was developed provides hardware redundancy using selective TMR. TMR was only applied to FPGA circuit components that were identified to be most susceptible to SEUs, using a specialized hardware tool. Software redundancy was provided using an enhanced version of SWIFT called SWIFT-R that triplicated data and instructions. As was seen with the hardware method, SWIFT-R was selectively applied to a subset of instructions that use specific critical registers. The specialized software hardening tool developed could not only apply SWIFT-R, but could also provide estimated fault-coverage results using only the applied software redundancy method. The architecture was tested

by simulating the performance of an 8-bit PicoBlaze microprocessor with the hybrid architecture applied. The simulations that were performed explored the trade space between full implementation of selective TMR paired with SWIFT-R, and a non-hardened processor. The results highlighted that the full redundancy implementation provided 100% reliability, but with the highest overhead, where some intermediate redundancy implementations demonstrated a better reliability/overhead relationship [14].

Hybrid redundancy methods generally present greater design complexity, but provide an optimized design solution while considering performance, cost, and reliability. Applications instantiated on SRAM FPGAs intended for use in space are subject to strict Size, Weight, Power, and Cost (SWAP-C) constraints, and thus hybrid redundancy methods are most suited towards these systems. However, hybrid redundancy methods are typically static methods, which are designed to operate under a specific radiation profile. Even greater performance gains and reliability robustness can be achieved using a form of hybrid-redundancy that can respond dynamically or adapt to the current operational environment, as was seen with the adaptive spatial redundancy methods described in Section 2.4.1.3.

Adaptive Hybrid Redundancy (AHR) [20] was designed and developed to offer a unique radiation mitigation strategy unlike those previously discussed, that provides flexibility in terms of system performance and reliability. This research is focused on further maturing AHR and experimentally validating its performance as applied to a MIPS processor implemented on an FPGA. The design of AHR MIPS and simulation results of its performance are described in-depth in Section 2.6. However, before discussing AHR further background is necessary on the ways in which radiation mitigation strategies are tested to determine their efficacy.

2.5 V&V of Radiation Mitigation Methods for SRAM FPGAs

To appropriately verifiy the performance of an SRAM FPGA that uses a radiation mitigation methodology requires either physical testing and/or testing through simulation.

2.5.1 Physical Radiation Testing

Radiation testing is a necessary step to determine the TID and SEU characteristics of an FPGA device. TID characterization is performed by exposing the target FPGA to a constant radiation source, such as Cobalt-60 [13]. TID performance of a device is mostly determined by the design and manufacture of the device, and thus the TID performance for a given FPGA device is usually available through the manufacturer or through membership to a radiation test consortium.

Unlike TID, SEU testing must usually be independently performed on the FPGA with the implemented design of interest. Typically, SEU characteristics are tested by exposing the target FPGA to high-energy particles produced by a generator or accelerator [9, 47]. SEU testing can be executed in two different ways: static testing and dynamic testing [28, 41, 47]. The static testing strategy involves continuous readback of the FPGA Device Under Test's (DUT's) configuration bitstream while being subjected to radiation, which is then compared against an error-free copy to count the number of bit-flips that occur during testing [5, 28]. The dynamic testing strategy focuses on observing the output of a test program executed by the target FPGA. The test program is used to activate sensitive portions of the design, which are then subjected to radiation. The dynamic testing strategy is not as comprehensive as the static strategy, but can still be used to determine the performance of the implemented mitigation strategy [28, 42]. The results of both static and dynamic testing can be used to determine the register-bit cross-section, or SEU cross-section,

and application error rate [42]. The SEU cross-section and application error rate are necessary to characterize the performance of the target FPGA design/device.

Radiation testing can approximate the space-radiation environment and is an important step towards maturing development and readiness of a circuit design on an FPGA. However, radiation testing is expensive to perform and the current demand for testing has led to long-lead times for scheduling testing time [9]. The scheduling constraints caused by radiation testing are not ideal for space programs, where it is beneficial to perform analysis of SEU effects on the target design early to determine feasibility and identify vulnerabilities [57]. For these reasons, methods have been developed to characterize the radiation performance of a design by simulating radiation effects.

2.5.2 Fault Injection Campaigns

Fault injection refers to the random injection of bit flips in both time and location on the target device while executing a test or benchmark program. A method of fault injection used to simulate an SEUs effect on an FPGA memory cell is the Code Emulating an Upset (CEU) method [41]. CEU involves injecting a bit-flip into a random memory cell of the DUT that is running a test application. The output of the application is monitored to determine the effect of the randomly injected bit-flip. By executing enough CEU tests, a statistically valid result can be obtained for the number of faults required to produce a specific type of error in the FPGA DUT [41]. CEU tests are used to perform SEU characterization of a redundancy strategy implemented on an FPGA. The execution of many CEU tests for characterization purposes is referred to as a fault injection campaign. Carefully constructed and thorough fault injection campaigns can provide SEU characterization results with nearly negligible error rates as compared to results obtained through radiation testing [42].

Fault injection campaigns also need to quantify the effects of radiation induced errors produced in the configuration memory of an FPGA. Fault injection in configuration memory usually requires the use of two FPGA devices. One FPGA is used as the error-free reference, and the other is subjected to configuration memory errors. Configuration memory bit-flips are introduced to one of the FPGAs by modifying the bitstream used to configure that device. The operation of each device is then compared to determine the effect(s) of the injected configuration error. Each bit in the bitstream is modified and tested in this way, to determine the effects of an SEU on every configuration cell in the FPGA design. A new bitstream must be generated and used to re-configure the FPGA DUT, and it is for this reason that configuration memory fault injection campaigns can take a significant amount of time to perform [28]. Many automated or optimized fault injection methods have been developed to reduce the amount of time required to perform a fault injection campaign on the configuration memory of an FPGA design [15, 30, 49].

This research uses a specialized hardware module to perform the CEU method of fault injection and is described in Section 2.6.5.1. The fault injection campaign executed for this research does not include tests on configuration memory and does not perform a full factorial injection campaign on memory elements used by the test application. The fault injection campaign was de-scoped in this manner, because only specific error scenarios needed to be tested and due to time constraints.

2.6 Adaptive Hybrid Redundancy (AHR)

The AHR architecture was previously developed by Hamilton in [20], and that work is the foundation upon which this research is built. The design and testing performed to verify that AHR is a viable redundancy method in [20] is technically dense, because it involves the design of many custom hardware components, including

a customized processor that could utilize AHR. This section will present the aspects of the research in [20] that were directly used to accomplish the research presented in Ch. 3, and because of this some aspects of the original AHR design will not be discussed in-depth.

AHR is a hybrid hardware & software redundancy scheme that was applied to a custom MIPS architecture defined in Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). The AHR MIPS architecture consists of three basic MIPS processors, a voter module, an AHR controller, and memory. The AHR architecture design combines the TMR hardware design philosophies discussed in Section 2.4.1.1, and the EDDI - TSR software redundancy method discussed in Section 2.4.2.1. A diagram of this architecture is shown in Figure 2.

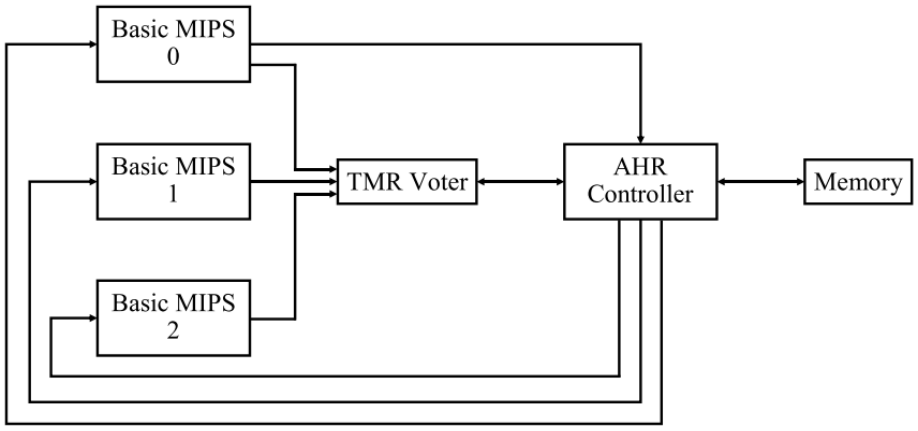

Figure 2. AHR MIPS Simplified Block Diagram

The AHR MIPS architecture can operate in either a TMR or TSR mode. In TMR mode the architecture actively uses all components shown in Figure 2. In TSR mode the architecture does not use the TMR Voter and only uses one of the three available basic MIPS processors. Section 2.6.1 will discuss the design and purpose of each component in the AHR MIPS architecture.

2.6.1 AHR MIPS Components

2.6.1.1 Basic MIPS Processor

The AHR architecture was designed to provide redundancy to a MIPS processor system. The AHR architecture uses three Basic MIPS processors while operating in TMR mode and only uses one Basic MIPS processor while operating in TSR mode. The Basic MIPS processor is a custom built component that implements a 33 instruction subset of the MIPS32® architecture [31]. This instruction subset is listed in Table 8 in Appendix A. The Basic MIPS processor performs operations sequentially and does not implement a pipelined or super-scalar design. The processor was designed to only process a subset of MIPS instructions and be sequential, because implementing a feature rich and complex MIPS processor would have detracted from the goal of the research. A simple sequential MIPS processor that used a subset of the MIPS instruction set was efficient to implement, and was all that was necessary to determine the effectiveness of AHR for mitigating and protecting a device against radiation induced errors.

The Basic MIPS processor is split into a Controller and Datapath. The Controller consists of a controller Finite State Machine (FSM), an instruction register, a module that generates state dependent control signals, and an instruction decoder/encoder that provides the appropriate data and control signals to the Datapath [20]. As a high-level summary, the operation of the Controller is dictated by the controller FSM module. The controller FSM module has 10 states and uses information about the current state and the type of instruction being executed to iterate through those 10 states. The controller FSM uses an output from memory called the MEM_READY signal to determine when memory has completed a read or write operation. The instruction decoder/encoder module informs the controller FSM module what type of instruction needs to be processed. The Controller begins processing an instruction

once MEM_READY is high, which indicates that memory is ready to perform a new operation. An instruction is then fetched from memory and decoded by the instruction decode/encode module. The instruction decode/encode module outputs data signals to the Datapath to perform the mathematical operation necessary to process that instruction and also outputs a control signal to the controller FSM. The outputs of the controller FSM change depending on what type of instruction is being executed, as indicated by a control signal provided by the instruction decode/encode module.

The Datapath component contains a bank of 32-32-bit registers, a program counter, an Arithmetic Logic Unit (ALU), and multiplexers used to control data flow between Datapath components [20]. As a high-level summary, the operation of the Datapath is dictated by the control signals and decoded instruction received from the Controller. The data used by instructions and the results of those instructions are stored in the General Purpose Register (GPR) bank. The Datapath modifies the contents of the GPR bank depending on the instruction processed and can also modify the contents of memory.

The Basic MIPS processor designed by Hamilton in [20] was used without modification in this research, since this research seeks to take the AHR MIPS architecture and implement it in hardware.

Detailed information on the design of the Basic MIPS processor can be found in the Basic MIPS Architecture Technical Report by Hamilton [22].

2.6.1.2 TMR Voter

In the AHR MIPS architecture, the TMR Voter is the component that decides what the output of the three Basic MIPS processors to the AHR Controller and by extension memory should be. The TMR Voter component functionally operates in the same way as a majority voter described previously in Section 2.4.1.1. The TMR Voter

receives a READ, WRITE, 32-bit ADDRESS, and 32-bit MEM IN/DATA signal from each of the three MIPS processors. The TMR Voter also receives a MEM_READY signal and a 32-bit data signal provided by the AHR Controller and by extension memory. The TMR Voter is implemented as an FSM, whose operation is dependent upon all input signals except for the input 32-bit data signal from the AHR Controller. The voter FSM was designed to not only check that all incoming signals from the MIPS processors are equivalent, but also that they arrive at the same time.

The voter FSM includes states that compare the control signal and data inputs provided by each Basic MIPS processor. Under normal operating circumstances, the voter FSM iterates through a series of error free states if the control and data inputs of all processors are determined to be equivalent or in agreement. The voter FSM iterates through distinct error states depending on whether one processor or every processor produce control or data inputs that are not in agreement. The overall behavior of AHR MIPS as a result of the TMR Voter operating under error free conditions or after an error is detected is described in Section 2.6.3.1.

The voter FSM also includes states that determine whether the control signal and data inputs provided by each MIPS processor arrive at nearly the same time. In the ideal case, each connected processor is supplied the same instructions at the same time, and therefore are expected to produce equivalent responses that appear at the input of the TMR Voter at the same time. However, in any complex logical system there are expected to be propagation/line delays that could cause signals expected to appear at the same time, to arrive at slightly different times. The voter FSM accounts for propagation delays by using a pre-defined TIMEOUT value. Whenever the voter FSM expects signals from the MIPS processors, a counter begins and if the counter value exceeds the TIMEOUT value, then the processor(s) that did not supply the expected signal are assumed to have experienced an error. The voter FSM

iterates through the same error states whether an error was detected in data or the TIMEOUT value was exceeded.

Traditional majority voters are only used for error detection and notify some other component to execute error recovery operations after an error is detected. The TMR Voter component used in the AHR MIPS architecture was designed to perform all the operations of a traditional majority voter while also performing new functions. These additional functions are that the TMR Voter needed to manage/create Save Restore Points (SRPs) and perform error recovery when a discrepancy was observed between the inputs provided by the processors.

The TMR Voter creates SRPs after a pre-defined number of instructions are processed without error, this pre-defined number is referred to as the SAVE POINT. The voter FSM will iterate through specific SAVE states once the SAVE POINT is reached. As mentioned in Section 2.6.1.1, each Basic MIPS processor has a 32 register GPR bank: 31 user registers, one of which contains the current program loop count, and one immutable zero register. As a high-level summary, once a SAVE POINT is reached the TMR Voter will instruct each processor to store each of its 31 registers and the current PC value to a specified location in memory. The TMR Voter can store a copy of those registers to one of two allocated locations in memory, one representing SRP0 and the other SRP1. The block of memory chosen is dependent on a pointer value that is held in memory directly following the blocks allocated for the SRPs. During both SRP creation and error recovery operations, the TMR Voter will load the SRP pointer from memory and PC value at which that special process began into its own registers. The retrieved pointer value indicates which SRP was most recently made, referred to as the active SRP, whereas the older SRP is referred to as the inactive SRP. When a new SRP needs to be made, the voter determines which SRP is inactive and incrementally writes the 31 registers and PC value, retrieved at

the beginning of SRP creation, to the portion of memory that corresponds to that SRP. The pointer value itself is used as an address offset to write the 31 registers and PC value to the block of memory that corresponds to the SRP being made. After the 31 registers and PC value have been written to the new SRP location, the voter FSM will update the pointer value location in memory to indicate the new active SRP. The voter FSM will then issue a branch command to each processor to return to the PC value at which SRP creation began, and execution of program instructions resumes.

The TMR Voter also needs to perform error recovery when discrepancies are found between the inputs provided by the processors. When an error has been found in only one of the three MIPS processors, the voter FSM performs single processor error recovery or Type A error recovery. The voter FSM iterates through the Type A error recovery states which begin by resetting the processor in error and storing the PC value at which the error was detected (supplied by the two processors in agreement). The voter then issues store word (SW) instructions to the processors in agreement, which both present the first of the 31 registers (R1) to the voter. The voter does not store R1 to memory, but instead issues a load word (LW) to the reset processor to load R1 into its GPR bank. This process repeats for all 31 registers. Once all the Type A error voter FSM states have been completed, all of the processors hold the same data, and the voter instructs each processor to branch to the PC value at which the error was detected and continue program execution. During Type A error recovery, the TMR Voter does not use either of the SRPs created during program execution. SRPs are only used when the voter identifies that none of the processors agree.

When none of the processors agree, the voter FSM has to perform multi-processor error recovery or Type B error recovery. If a multi-processor error occurs, the TMR Voter can no longer determine which processor is correct and requires that all pro-

cessors revert to a previous state held in the active SRP. To perform Type B error recovery, the voter FSM iterates through Type B error recovery states which begin by resetting all the processors. Each of the processors are issued LW instructions that copy the SRP register values held in memory to their own GPR banks. The active SRP is used to provide each register with the most up-to-date values from the corresponding location in memory. Once all 31 registers have been loaded, the TMR Voter loads the PC value at which the SRP was written, from memory, and issues a branch instruction to each processor to resume processing program instructions from that point.

The TMR Voter component designed by Hamilton in [20] was used without modification in this research since this research seeks to take the AHR MIPS architecture and implement it in hardware.

Detailed information on the design of the TMR Voter can be found in the Triple Modular Redundancy MIPS Architecture Technical Report by Hamilton [23].

2.6.1.3 AHR Controller

The AHR Controller component determines when the AHR MIPS architecture transitions from operating in one mode to the other, whether it be the transition from TMR to TSR or vice versa. The AHR Controller is implemented as an FSM and includes all the states necessary to seamlessly orchestrate the transition between modes. The AHR Controller component sits between the Basic MIPS processors, TMR Voter, and memory. The AHR Controller generates control signals and inputs to different multiplexers that control how signals are distributed between the Basic MIPS processors, TMR Voter, and memory. While in TMR mode, the Basic MIPS processors are connected to the TMR Voter and the AHR Controller routes signals from the TMR Voter to memory through the multiplexers. While in TSR mode, the

AHR Controller routes signals from one Basic MIPS processor to memory through the multiplexers, while holding the other two processors and the TMR Voter in a low power reset state. During a transition from one operating mode to another, the AHR Controller uses the multiplexers to route signals between the Basic MIPS processors, TMR Voter, and memory as necessary to ensure a seamless transition while also preparing the system for operation in the new mode. The design and operation of the AHR Controller is more easily discussed by putting its operation into context. The various ways in which the AHR Controller operates is described in Section 2.6.3.

Hamilton assumes in [20] that the AHR Controller and multiplexers used for signal routing are radiation hardened, therefore no redundancy methods were integrated into the design of those components. The AHR Controller component designed by Hamilton in [20] was modified for use in this research. The modifications made to the AHR Controller are described in Ch. 3 Section 3.3.5.1. These modifications only slightly modify the behavior of the controller to enable testing in hardware.

Detailed information on the design of the AHR Controller can be found in the Adaptive-Hybrid Redundancy MIPS Architecture Technical Report by Hamilton [21].

2.6.1.4 Memory

The memory component of the AHR MIPS architecture contains the AHR MIPS program that will be executed. As a result, a new memory module must be generated for each unique AHR MIPS program. The composition of an AHR MIPS memory module is determined by the AHR MIPS instruction set/program it contains. The structure of an AHR MIPS instruction set is described in Section 2.6.2.

The inputs to memory are a READ and WRITE signal, a 32-bit ADDRESS signal, and a 32-bit data signal (MEM_IN). The outputs are a memory ready signal (MEM_READY), a 32-bit memory data signal (MEM_OUT), and a PROGRAM

DONE signal. The memory component receives the READ and WRITE command signals from the AHR Controller, which determine how memory uses the provided ADDRESS and DATA IN signals. If the READ signal is high, memory will access the address specified by the ADDRESS signal and will output that data on the MEM_OUT signal. The output MEM_READY signal will be set high to indicate that the operation is complete and that the data on MEM_OUT is ready to be retrieved. If the WRITE signal is high, memory will write the MEM_IN data to the specified ADDRESS. Memory will then set the output MEM_READY signal high to indicate the write operation is complete.

Hamilton assumes in [20] that the memory module is radiation hardened, therefore no redundancy methods were integrated into its design. This memory design was not modified as part of this research, however new memory modules were generated using the same methods used by Hamilton [20] and described in Section 2.6.2.

Detailed information on the design of the AHR memory can be found in the Adaptive-Hybrid Redundancy for Radiation Hardening dissertation by Hamilton [20].

2.6.2 AHR MIPS Instruction Sets/Programs

The AHR MIPS architecture processes custom generated AHR instruction sets. An AHR instruction set is comprised of a set of TMR instructions, a set of TSR instructions which are functionally equivalent to the TMR set, temporary and permanent memory, TSR MIPS SRP creation instructions, TSR MIPS error recovery instructions, and SRP memory [20]. The TMR portion of an AHR instruction set is a standard un-modified version of the program that is executed while AHR MIPS operates in TMR mode. The TSR portion of an AHR instruction set is a modified version of the program which uses the EDDI software redundancy method discussed in Section 2.4.2.1. Table 3 provides an example of an AHR instruction set.

Table 3. Example AHR MIPS Instruction Set/Program [20]

Line Numbers	Description
1 - 49	TMR MIPS Program
50 - 152	TSR MIPS Program
153 - 161	Temporary and Permanent Memory
162 - 268	TSR MIPS SRP Creation Instructions
269 - 344	TSR MIPS Error Recovery Instructions
345 - 409	SRP Memory

Hamilton developed a set of MATLAB scripts to randomly generate instruction sets for TMR, TSR, and AHR MIPS as part of the research in [20]. Each generated instruction set is formatted with respect to the architecture by which it will be processed, where an example of an AHR MIPS instruction set was given in Table 3. The instruction sets are composed only of the 33 MIPS instructions listed in Table 8 of Appendix A. The MATLAB scripts allow a user to specify the number of instructions in a program, the number of variables, and other parameters to define the composition of the generated instruction sets. The MATLAB scripts ensure that the generated instruction sets will not produce an error due to illegal operations. The method used to generate random instruction sets by Hamilton was used to generate instruction sets for this research.

2.6.3 AHR MIPS Operational Modes

The AHR controller determines whether the AHR MIPS architecture operates in TMR or TSR mode. In TMR mode the controller uses all three processors and the voter module to process instructions supplied by memory. The AHR controller

continuously supplies new instructions to each of the processors and accepts incoming data supplied by the TMR Voter. In TSR mode the controller only uses output from one of the processors to process instructions supplied by memory, while the other two processors and the TMR Voter are held in a reset state. The AHR controller controls the state and operation of the AHR MIPS system, and performs different functions when errors are detected in each mode.

2.6.3.1 AHR MIPS - TMR Mode

The TMR operating mode is used as a high-performance, enhanced-redundancy operating mode. The MIPS processors in TMR mode execute the un-modified TMR portion of the AHR program which does not contain any additional duplicated instructions or branches, as seen in the TSR portion. Without the need to process additional instructions, the overall system runtime performance in TMR mode is the highest available in AHR MIPS. In addition, TMR mode provides the fastest error detection and correction capability, because single errors can be detected and corrected without the need to revert to an SRP. TMR SRPs are created during the execution of an AHR program. The TMR Voter initiates SRP creation after a user-specified number of instructions have processed without error. AHR MIPS operating in TMR mode uses two distinct SRPs stored in memory. Two SRPs are used in case an error is detected while writing one SRP, so that the other can be used to reload the system to a known state. The TMR Voter and AHR controller work together to dictate how the AHR MIPS architecture operates under normal conditions and also how the architecture detects and corrects any errors during runtime.

In an error-free scenario, the voter presents data that was equivalently produced by all three processors to the AHR Controller. In this scenario, the AHR controller simply passes data between memory and the TMR Voter, which is then distributed

to the Basic MIPS processors.

The AHR Controller is notified when the TMR Voter detects an error by setting a TMR error signal between the voter and controller high. Only the TMR Voter can detect errors while AHR MIPS operates in TMR mode. The TMR Voter detects discrepancies between the three connected Basic MIPS processors anytime the processors attempt to read from or write to memory.

The processors perform read operations whenever a new instruction is fetched from memory and while executing a LW instruction. During a read operation, the TMR Voter determines whether all three processors are attempting to read from the same address in memory or stated another way, have the same Program Counter (PC) values. The TMR Voter provides protection against program flow errors/illegal branches by continuously ensuring all processors have synchronized PC values.

The processors only perform a write operation while executing a SW instruction. During a write operation, the TMR Voter determines whether all three processors are attempting to write the same data from their GPRs to the same address at the same time. The TMR Voter ensures that a specific register in the GPR bank of each processor are equivalent by comparing them while executing SW instructions. A register in the GPR bank of all three processors is only checked for equivalency during SW instructions, because it would be inefficient to check every intermediate instruction for an error, when any radiation induced error will propagate until a request is made to write that final value to memory.

In an AHR TMR Type A error scenario, the voter will conduct TMR Type A error recovery and will keep the error signal to the AHR Controller high to indicate that the detected error is still being corrected. AHR MIPS in TMR mode performs Type A error recovery by copying the data contained in the two processors in agreement over to the processor that presented incorrect data in the same manner as described

in Section 2.6.1.2. This ensures that all the processors contain the same data before resuming program execution. Once error recovery is complete the TMR Voter will set the error signal low, indicating to the AHR Controller that an error has been corrected and normal program execution has begun.

In an AHR TMR Type B error scenario, the voter determines that none of the attached processors produced data equivalent to one another. The voter will set the error signal to the AHR Controller high and will execute TMR Type B error recovery operations. AHR MIPS in TMR mode performs Type B error recovery by pausing program execution, and using an SRP stored in memory to restore each processor back to a previous state in the same manner as described in Section 2.6.1.2. After each processor has been loaded with the values held in SRP memory, the voter will set the error signal to the AHR Controller low, and the AHR MIPS architecture resumes normal program execution.

AHR MIPS transitions from TMR mode to TSR mode only after a specified number of TMR instructions have been processed without error. The number of error free instructions that need to be processed before transitioning is called the TMR to TSR Transition Point. A counter in the AHR Controller is used to track how many instructions have been processed without error. If this counter reaches the TMR to TSR Transition Point, then a transition can be executed. Anytime an error occurs in TMR mode this counter is reset. The other transition condition that must be met to move to TSR mode, is that the TMR program be at the beginning of a program loop. The AHR program performs the TMR to TSR transition by splitting user-data register space at that point in time into two distinct sets. These sets will hold the data produced while processing the TSR portion of the AHR program, and are compared against each other during runtime in accordance with EDDI TSR [36]. By performing the TMR to TSR transition at the beginning of a loop, the only value that must be

maintained during and after the transition is the loop count. All other data will be re-calculated because the program begins execution at the beginning of the TSR portion of the AHR program, and despite taking longer the TSR portion will produce the same results and interact with memory in the same way as the TMR portion. Once the transition has completed, the AHR MIPS architecture begins executing the TSR portion of the AHR program. The AHR MIPS architecture operates in TSR mode until either the AHR MIPS program is completed or a specific error scenario occurs.

2.6.3.2 AHR MIPS - TSR Mode

The TSR operating mode is used as a low-power operating mode with reduced processing performance. While in TSR mode, the AHR Controller holds two out of the three processors in reset and only accepts data coming out of the last remaining processor. AHR MIPS in TSR mode must process a little over 2x as many instructions as AHR MIPS in TMR mode. By processing more instructions, the system performance is reduced as compared to operation in TMR mode. However, because less of the FPGA fabric is used to operate AHR MIPS in TSR mode, the power utilization is less than AHR MIPS in TMR mode. AHR MIPS operating in TSR mode is not as quick to recover from errors as in TMR mode because a single error detected in TSR mode requires that the processor revert to a previous state by loading from an SRP. Also, TSR mode is not as resilient as TMR mode because the selected method of TSR provides no protection against control flow/illegal branching errors.

TSR SRPs are created at specific loop counts during program execution. The loop counts at which TSR SRPs are made are set by the user and included as instructions in the TSR program. AHR MIPS operating in TSR mode uses two distinct SRPs stored in memory to provide an additional backup system state in case an error is

detected while writing one of the SRPs. TSR mode SRP creation is dictated by a set of instructions and writes each register-pair to the inactive block of TSR SRP memory, which is determined by inspecting the TSR SRP location pointer. The TSR SRP creation process is similar to TMR mode SRP creation, except that there are less registers to write to memory and the TSR SRP location pointer is loaded into a register in the active processor, for use as a memory index while executing the TSR SRP creation instructions.

In error-free and error scenarios, the AHR Controller only interfaces with one processor and memory. AHR MIPS in TSR mode can only detect errors just before a SW instruction is processed. The TSR portion of the AHR program adheres to the concepts set by EDDI TSR [36], which uses a BNE instruction inserted before a SW to detect an error in the register pair of interest. If the data in the register pair matches, the program continues execution and the agreed upon value is stored to memory. If the BNE instruction determines that the values in the register pair are not equivalent, then the program jumps to the TSR error recovery section of the AHR program which executes a system reload from the active SRP. TSR mode error recovery is similar to TMR mode Type B error recovery, except that in TSR mode, the process is executed using instructions rather than iterating through a series of states defined in a FSM like the TMR Voter, and the TSR SRP location pointer is loaded into a register in the active processor, for use as a memory index while executing the error recovery instructions.

AHR MIPS transitions from TSR mode back to TMR mode when two consecutive errors are detected while executing the same portion of the TSR program. As described previously, a single detected error in TSR mode requires that the processor revert to a previous state using an SRP. After re-loading from an SRP, a flag is set which indicates that an error occurred. This flag is cleared after creating the next

TSR SRP, which occurs at specific points during program execution. If the flag is set high and another error is detected before writing the next TSR SRP, then the AHR Controller will execute a TSR to TMR transition in order to maintain acceptable processor performance. During a TSR to TMR transition, the AHR Controller begins by holding all processors in reset. The controller will then access the active TSR SRP and load the loop count held in that SRP into the appropriate loop counter locations of the TMR SRPs memory. These loop counts are decremented by one because after transition AHR MIPS will resume operation at the beginning of the TMR program rather than the end where TMR decrements the loop counter. The controller also loads each TMR SRP with the PC address that corresponds with the start of the TMR program and sets the active TMR SRP to SRP0. After properly configuring the TMR SRPs, the AHR Controller will stop sending the reset signal to the three Basic MIPS processors and will instruct each processor to load the loop counter to begin TMR program execution.

Despite being included in the design of the AHR Controller, the TSR to TMR transition process is not explicitly included in AHR MIPS performance evaluations by Hamilton in [20]. The TSR to TMR transition was verified in simulation, but was not included during performance evaluations because it was determined that the anticipated on-orbit error rate is low enough that it is unlikely that the criteria necessary to perform the TSR to TMR transition would be met. The criteria to perform the TSR to TMR transition may only be met during cosmic or man-made events that increase the radiation level of the operational environment. As a result, the TSR to TMR transition was not tested in this research.

2.6.4 Evaluation of Error Free AHR MIPS

After developing the AHR MIPS architecture shown in Figure 2, which operates as described in Section 2.6.3, the function of the architecture was verified through simulation and its performance was evaluated.

2.6.4.1 Simulation of Error Free AHR MIPS

AHR MIPS was functionally verified by first simulating the operation of both a standalone TMR MIPS architecture and TSR MIPS architecture. An incremental approach was taken to ensure that AHR MIPS would operate correctly, by simulating each architecture that it operated in separately. Every component of TMR, TSR, and AHR MIPS were written in VHDL, and were functionally verified using the Mentor Graphics QuestaSim software [20]. A basic program was run on TMR MIPS and TSR MIPS to see that the architectures could properly execute their respective programs and were correctly creating SRPs. Once it was shown that TMR and TSR MIPS were functioning correctly, AHR MIPS was functionally verified. The operation of AHR MIPS in each mode was correct, mode specific SRPs were correctly created, and the TMR to TSR transition was correctly executed.

This incremental approach for functional simulation was also used in this research, because it allows for errors to be more easily identified and addressed at lower levels before integration into higher level components.

2.6.4.2 Timing of Error Free AHR MIPS

After functional verification was complete, equations were created to calculate the error-free runtime of any program processed on its respective architecture. The error free timing equations were created after measuring the time it took to process basic instructions, branch instructions, SW/LW instructions, perform SRP creation,

and various other processes using simulations. In the context of timing, "operations" will be used to refer to basic instructions, branch instructions, SW/LW instructions, perform SRP creation, and various other processes. With knowledge on how long each of the previously listed operations takes to perform, the runtime of any error free program could be calculated after determining how many instructions of each type were in a program, and how many SRPs would be created. The 33 MIPS instructions measured are listed in Table 8 in Appendix A.

All of the error free timing equations created by Hamilton [20] and discussed in this section are used and/or modified for use in this research presented in Ch. 3 Section 3.4.2. All of these equations are written in MATLAB.

The runtime of an error free TMR MIPS program is calculated using Equation 1 [20] where $T_{TMR\ MIPS}$ is the time to complete the entire TMR program, $T_{TMR\ init}$ is the time to initialize the loop counter of the program which only has to occur once per program, $T_{TMR\ loop}$ is the time it takes to complete one loop of the program, n_{SRP} is the number of TMR SRPs that are made during the program, $T_{TMR\ SRP}$ is the time it takes to create a TMR SRP, and $T_{TMR\ conc}$ is the time to complete the TMR program after the last loop. Equation 2 [20] calculates $T_{TMR\ loop}$ by summing the time it takes to complete every instruction in one program loop where N_{TMR} is the total number of program instructions and $t_{I_{TMR\ n}}$ is the time it takes to process instruction n. Equation 3 [20] calculates n_{SRP} where $n_{TMR\ init}$ is the number of instructions to initialize a TMR program, n_{loops} is the total number of program loops, $n_{TMR\ conc}$ is the total number of instructions processed after the final loop has been processed, and n_{save} is the number of instructions that need to be processed before creating an SRP.

$$T_{TMR\ MIPS} = T_{TMR\ init} + n_{loops} \cdot T_{TMR\ loop} + (n_{SRP} - 1) \cdot T_{TMR\ SRP} + T_{TMR\ conc} \tag{1}$$

$$T_{TMR\ loop} = \sum_{n=1}^{N_{TMR}} t_{I_{TMR\ n}} \tag{2}$$

$$n_{SRP} = \left\lfloor \frac{n_{TMR\ init} + N_{TMR} \cdot n_{loops} + n_{TMR\ conc}}{n_{save}} \right\rfloor + 1 \tag{3}$$

The runtime of an error free TSR MIPS program is calculated using Equation 4 [20] where $T_{TSR\ MIPS}$ is the time to complete the entire TSR program, $T_{TSR\ init}$ is the time to initialize the loop counter of the program which only has to occur once per program, $T_{TSR\ loop}$ is the time it takes to complete one loop of the program, $T_{TSR\ SRP0}$ is the time it takes to create SRP0, $T_{TSR\ SRP1}$ is the time it takes to create SRP1, $T_{TSR\ conc}$ is the time to complete the TSR program after the last loop, and $T_{TSR\ skip}$ is the time it takes to perform the 6 instructions that determine when the 3 TSR SRPs are created.

Equation 5 [20] calculates the time required to complete a single error free TSR program loop, where N_{TSR} is the total number of instructions in one TSR program loop and $t_{I_{TSR\ n}}$ is the amount of time it takes to process instruction number n of the program. The TSR programs used during the research in [20], and in this research, always create a total of 3 SRPs during a given program. SRP0 is initialized at the beginning of a TSR program, so the first SRP created in the program is SRP1, then SRP0, and finally SRP1 again. This is why $T_{TSR\ SRP1}$ is multiplied by 2 and $T_{TSR\ SRP0}$ is not in Equation 4. There are a total of 6 program instructions that are used to determine if an SRP should be made, where one pair determines when the first SRP is made, the second pair determines when the second SRP is made, and the third pair determines when the third SRP is made. A TSR SRP is made by branching to the portion of the instruction set that contains SRP creation instructions and processing those instructions. Once those instructions are completed, the program branches back

to the main program at the next instruction after the 6 that determine when an SRP is made. Due to these branching characteristics, by the end of a TSR program each pair of SRP create instructions will be skipped once. $T_{TSR\ skip}$ is subtracted from the total TSR runtime in Equation 4 to ensure that the skipped instructions are properly taken into account.

$$T_{TSR\ MIPS} = T_{TSR\ init} + n_{loops} \cdot T_{TSR\ loop} + T_{TSR\ SRP0} + 2 \cdot T_{TSR\ SRP1} + \cdots \\ T_{TSR\ conc} - T_{TSR\ skip} \tag{4}$$

$$T_{TSR\ loop} = \sum_{n=1}^{N_{TSR}} t_{I_{TSR\ n}} \tag{5}$$

An error free AHR MIPS program is executed in both TMR and TSR mode. Calculating the total runtime of an AHR MIPS program requires knowledge on how long the program was executed in each mode. The user specifies the number of instructions that must be processed to initiate the transition from TMR to TSR mode represented by $n_{transition}$, and therefore determines how many program loops are completed in each mode. Equation 6 [20] determines the number of loops that complete before transitioning called P_{loops} and is calculated by subtracting $n_{TMR\ init}$ from $n_{transition}$, because the divisor N_{TMR} accounts for all instructions in one TMR program loop, but this value does not include $n_{TMR\ init}$ which occur outside of the program loop. The ceiling is taken of the result because the TMR to TSR transition must occur at the beginning of a loop.

$$P_{loops} = \left\lceil \frac{n_{transition} - n_{TMR\ init}}{N_{TMR}} \right\rceil \tag{6}$$

The number of TMR SRPs created during a given AHR MIPS program (n_{CSRP}) was calculated using Equation 7 [20]. The n_{CSRP} can be found by determining how

many total TMR instructions are processed before transitioning to TSR mode, dividing that number by the number of TMR instructions that need to be processed before creating a TSR SRP (n_{save}), and taking the floor of that value.

$$n_{CSRP} = \left\lfloor \frac{n_{TMR\ init} + N_{TMR} \cdot P_{loops}}{n_{save}} \right\rfloor \tag{7}$$

The runtime of an error free AHR MIPS program was calculated using Equation 8 [20], where the runtime $T_{AHR\ MIPS}$ is found by determining how long the program ran in TMR mode ($t_{AHR\ TMR}$) and in TSR mode ($t_{AHR\ TSR}$). The instruction sets generated in [20] set the program loop count to be 999. Equation 8 uses an if statement to calculate the time spent in TSR operations. The $t_{AHR\ TSR}$ value is dependent on P_{loops} because TSR SRPs are created at user defined loop counts. If the transition point occurs early, $P_{loops} < 250$, then all TSR SRPs need to be made and the $t_{AHR\ TSR}$ value can be calculated directly using Equation 4. However, if the transition point occurs later in the program, $500 \leq P_{loops} < 750$, then only one TSR SRP is made before the program ends, and only two out of the six TSR SRP creation instructions are executed, thus requiring that $\frac{2}{3}T_{TSR\ skip}$ be subtracted to properly calculate $t_{AHR\ TSR}$.

$$
\begin{aligned}
&t_{AHR\ TMR} = t_{TMR\ init} + P_{loops} \cdot T_{TMR\ loop} + \cdots \\
&T_{TMR\ SRP} \cdot n_{CSRP} + t_{TMR \rightarrow TSR} \\
&if\ P_{loops} < 250 \\
&\quad t_{AHR\ TSR} = (n_{loops} - P_{loops}) \cdot t_{TSR\ loop} + T_{TSR\ SRP0} + \cdots \\
&\quad 2 \cdot T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip} \\
&elseif\ 250 \leq P_{loops} < 500 \\
&\quad t_{AHR\ TSR} = (n_{loops} - P_{loops}) \cdot t_{TSR\ loop} + T_{TSR\ SRP0} + \cdots \\
&\quad T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip} \\
&elseif\ 500 \leq P_{loops} < 750 \\
&\quad t_{AHR\ TSR} = (n_{loops} - P_{loops}) \cdot t_{TSR\ loop} + T_{TSR\ SRP1} + \cdots \\
&\quad T_{TSR\ conc} - \tfrac{2}{3} T_{TSR\ skip} \\
&elseif\ P_{loops} \geq 750 \\
&\quad t_{AHR\ TSR} = (n_{loops} - P_{loops}) \cdot t_{TSR\ loop} + T_{TSR\ conc} \\
&end \\
&T_{AHR\ MIPS} = t_{AHR\ TMR} + t_{AHR\ TSR}
\end{aligned}
\tag{8}
$$

After collecting the timing information and creating the above runtime equations, Hamilton predicted in [20] that TMR MIPS would have the fastest runtimes, followed by AHR MIPS, and then TSR MIPS. TSR MIPS was expected to take the longest because TSR programs contained twice as many instructions and interface with memory more frequently than the other architectures. TMR MIPS was expected to run the fastest because the TMR instruction sets are unmodified and do not incur the overhead seen with TSR MIPS. AHR MIPS was expected to have runtimes faster than TSR MIPS because AHR MIPS begins in TMR mode and processes a portion of the program quickly. However, AHR MIPS was expected to have runtimes slower than TMR MIPS because AHR MIPS does transition to TSR mode and takes longer

to complete the program.

2.6.4.3 Error Free Test Methodology

The runtime performance of each architecture was determined using the equations developed in Section 2.6.4.2 for a total of 1,000 randomly generated instruction sets. Each instruction set had three variants, where TMR MIPS used the Basic MIPS program, TSR MIPS used the TSR MIPS program, and AHR MIPS used the AHR program. Each variant of an instruction set produces the same results and store the same results to memory. The only difference between variants is how they are processed by their respective architecture. Hamilton [20] also determined the energy consumed by each architecture using separately developed power equations similar to those presented in Section 2.6.4.2. These equations are not included in this discussion, because energy calculations/measurements were determined to be out of scope for this research as will be discussed in Ch. 3 of this document.

40 additional instruction sets were generated for hardware in the loop (HITL) testing of each architecture. HITL testing was first conducted for the TMR MIPS and TSR MIPS architectures by programming a single Terasic SoCKit development board [52] with an Altera Cyclone V FPGA chip. A Keysight DSOS054A Infiniium S-Series Digital Storage Oscilloscope was used to measure the DONE signal produced at the end of program execution. The DONE signals were measured, and the program timings appeared to match those calculated, while also confirming the prediction that TMR MIPS would have the fastest runtimes as compared to TSR MIPS. However, using a single FPGA board for HITL testing was discovered to produce energy data that did not match calculations, and so a second HITL testing method was created.

The second HITL testing method split a given MIPS architecture across two Terasic DE10-Standard boards [54]: one containing the MIPS processor and the other

containing the corresponding memory module. This method was created to collect energy measurements of the processor independently from memory. Communication between the boards, or between processor and memory, was performed using a parallel communication scheme that assigned all 100 necessary signals to on-board GPIO pins. Each of the 100 signals was then transmitted to the other board using wires. This second HITL testing method was unable to implement any of the MIPS architectures, due to timing issues caused by implementing a parallel communication scheme. These challenges encountered by Hamilton during HITL testing are explained in more detail in Ch. 3 Section 3.2.1. The focus of the research presented in this document is to successfully implement error free TMR, TSR, and AHR MIPS in hardware by using a serial communication scheme to overcome the described HITL challenges.

2.6.4.4 Error Free Simulation/Calculation Results

Since HITL testing could not be performed, the performance results of each error free architecture were derived from simulation and calculations. Only a high-level summary of the results will be given, with emphasis on the runtime performance behaviors, which are the only results pertinent to this research. For an in-depth discussion of the results consult Adaptive-Hybrid Redundancy for Radiation Hardening [20] by Hamilton.

The results of the error free simulations/calculations for TMR, TSR, and most importantly AHR MIPS were determined to match predictions. Hamilton found that TMR MIPS completes programs faster than AHR MIPS, and AHR MIPS completes programs faster than TSR MIPS. Hamilton also found that TMR MIPS uses more energy than AHR MIPS, and AHR MIPS uses more energy than TSR MIPS. Hamilton was able to show that changing the TMR to TSR transition point allows AHR MIPS to have runtimes that fall between the maximum, set by only operating in TSR

mode, and the minimum set by only operating in TMR mode. This demonstrated the versatility provided by AHR to space system designers and mission planners, where AHR enables a MIPS processor to have flexible performance characteristics in terms of runtime and energy consumption determined by the TMR to TSR transition point.

These results are only for the architectures under error free operating conditions. The next portion of Hamilton's research effort [20] was to perform the same evaluation steps that were presented in Section 2.6.4, but for each architecture operating under various radiation induced error scenarios.

2.6.5 Evaluation of Error Prone AHR MIPS

The novelty of AHR MIPS was that it was designed to provide not only energy and runtime performance flexibility, but also provide variable radiation tolerance by either operating in TMR or TSR mode. Evaluation of AHR MIPS under various error scenarios was accomplished in the same way as was done in the error free case, where the performance of TMR MIPS, TSR MIPS, and AHR MIPS were all compared to contextualize the benefits provided by the AHR scheme.

2.6.5.1 Error Injection Module

To evaluate the performance of each architecture under various error scenarios required that simulated radiation induced errors be generated and injected into the architectures during operation. These simulated radiation induced errors were generated using an error inject module. A diagram of the MIPS datapath including the error inject module (shown in red) is given by Figure 3.

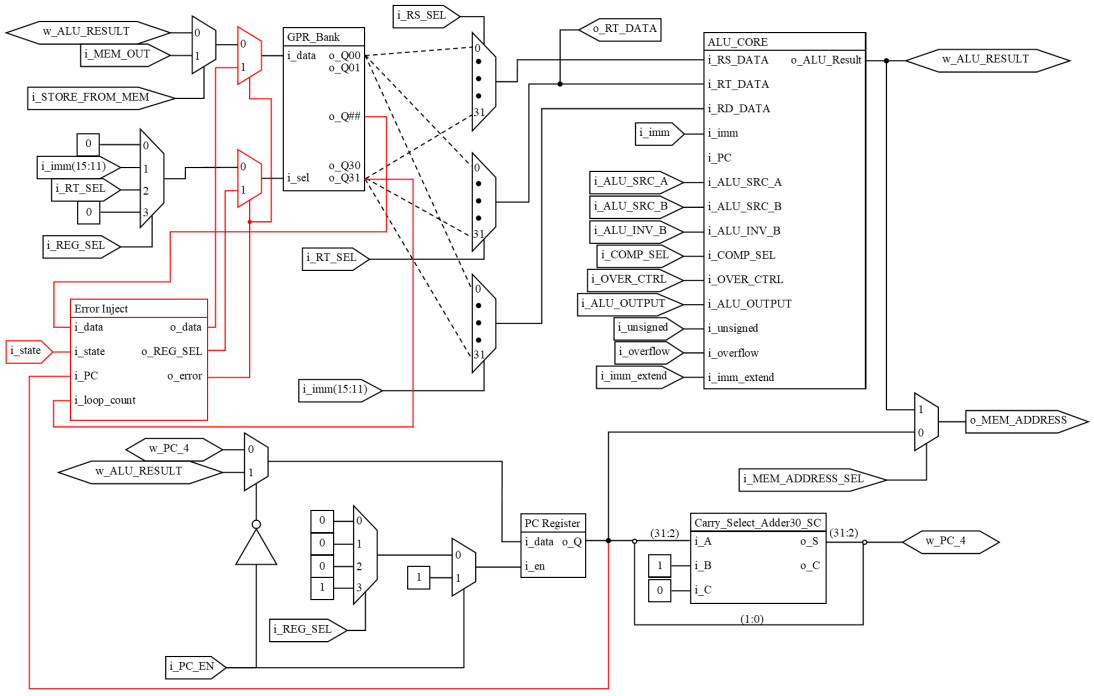

Figure 3. AHR MIPS Datapath w/Error Inject Module

The inputs to the module are a 32-bit data signal (i_data), a 4-bit state signal (i_state), a 30-bit PC signal (i_PC), and a 32-bit loop count signal (i_loop_count). The i_data and i_loop_count signals come from the GPR bank used by a Basic MIPS processor. The i_state signal is provided by the MIPS controller (not shown in Figure 3). The i_PC signal comes from the output of the PC register. The module is pre-configured to inject an error into a specific register at a specific PC value and loop count. Thus, the i_data signal is hardwired to the register that will be injected with an error, once the specified i_PC and i_loop_count are met. The i_state signal is used to ensure that the error is injected during state 0 of the MIPS controller, or before the MIPS processor begins processing the next program instruction. More details on the Basic MIPS controller FSM can be found in [22]. The error inject module

will inject a specified bit-flip error into the data on i_data and will only write that erroneous data (o_data) to the specified register so long as both the register select (o_REG_SEL) and the error signal (o_error) are set high. These signals are inputs to control multiplexers on the input of the GPR bank, and go high once the state, PC, and loop count conditions have been met. The error injector module was designed to have an interface compatible with TMR, TSR, and AHR MIPS. However, the point at which an error should be injected differs by architecture.

Datapath errors can only be detected at specific points during program execution, and these points differ depending on the MIPS architecture. As discussed in Section 2.6.3.1, TMR MIPS can only detect errors in user registers when an instruction that wants to write a register value to memory is processed. TMR MIPS can also detect PC value discrepancies when each processor retrieves an instruction from memory or performs a branch instruction. However, the error injector was used to only inject errors into user registers, and not registers that dictate the control flow of a program. Similarly, as discussed in Section 2.6.3.2, TSR MIPS uses comparison instructions to detect errors, which can occur in either one or both registers in a register pair. AHR MIPS processes an AHR program that contains both a TMR and TSR version of the program. AHR MIPS can either operate in TMR or TSR mode, and thus can only detect errors that are properly injected with respect to the mode that it is currently operating in. The differences in how each architecture detects errors in user registers required that the error injector be configured in slightly different ways to produce valid and detectable errors.

After design, the error inject module was simulated to perform functional verification. The error module was found to be compatible with each MIPS architecture and was able to properly inject errors into each. Using the error inject module, various new operational modes and error scenarios for each architecture were able to be

simulated.

The error inject module component designed by Hamilton in [20] was modified for use in this research. The modifications made to the error inject module are described in Ch. 3 Section 3.3.5.2. These modifications alter when the error is injected to account for specific edge cases.

2.6.5.2 Error Scenarios/Operational Modes of AHR MIPS

The error inject module was developed to simulate an SEU in the datapath of a Basic MIPS processor used in a given MIPS architecture. A Basic MIPS processor that incorporates an error inject module is referred to as an error-prone processor. The error performance of a MIPS architecture refers to the runtime of a program on that architecture which experiences an error during operation. To determine the effects of a single error in each architecture, required the use of one error-prone processor. To determine the effects of two independent errors in TMR and AHR MIPS, required the use of two error-prone processors. Only TMR and AHR MIPS were subjected to two errors because TSR MIPS and AHR MIPS operating in TSR mode perform the same error recovery operations whether one or two errors are detected.

All the error scenarios/operational modes tested for each MIPS architecture are listed in Table 5, where the results of operations in error free modes were previously discussed in Section 2.6.4.

Table 4. Architectures and Operational Modes Measured in Hardware

MIPS Architecture	Operational Mode
TMR	No Error (NE)
	Error A
	Error B - Best
	Error B - Worst
TSR	No Error (NE)
	Best Single Error
	Worst Single Error
AHR	No Error (NE)
	TMR Mode - Error A - Early
	TMR Mode - Error A - Late
	TMR Mode - Error B - Best Early
	TMR Mode - Error B - Best Late
	TMR Mode - Error B - Worst Early
	TMR Mode - Error B - Worst Late
	TSR Mode - Best Single Error
	TSR Mode - Worst Single Error

The new AHR MIPS architecture developed in this research operates in every mode listed in Table 5, and for this reason an in-depth description of each error mode and how they are realized is described in Ch. 3 Section 3.5.2.

2.6.5.3 Simulation of Error Prone AHR MIPS

Each MIPS architecture was slightly modified to incorporate the error injector module(s) in order to realize every error scenario. Each architecture was then functionally simulated in the same way presented in Section 2.6.4.1. Having shown that each error scenario could be produced in simulation, various new architecture specific processes were measured/timed to create the timing equations used to calculate the runtime of programs processed on the error prone MIPS architectures.

2.6.5.4 Timing of Error Prone AHR MIPS

After functional verification of each error prone MIPS architecture was complete, revised equations based upon those discussed in Section 2.6.4.2 were created. These new equations were used to calculate the runtime of any program processed on its respective architecture that experienced either an SEU or two simultaneous and separate SEUs. The timing equations were created by measuring the time it took to perform error detection and execute error recovery processes for each unique architecture using simulations. The error detection and recovery processes timed for each architecture are listed in Table 9 of Appendix A.

The error prone MIPS timing equations developed by Hamilton in [20] are generalized and can determine the runtime of an instruction set and can calculate the instruction and loop count where an error needs to be injected to realize each error scenario. All of the error prone equations developed by Hamilton in [20] were modified and used for this research, and because of the length and complexity of these equations, the discussion on how they were both developed and used is presented in Ch. 3 Section 3.5.2.

2.6.5.5 Fault Injection Test Methodology

To satisfy Hamilton's research goals [20], the performance of error prone TMR, TSR, and AHR MIPS in terms of program runtimes and energy were compared. As was done in Section 2.6.4.3, the error prone timing equations developed in Section 2.6.5.4 were used to calculate the program runtimes for each generated instruction set. The same 1,000 randomly generated instruction sets used to compare the error free performance of each MIPS architecture were used to compare the error prone performance of each MIPS architecture. Using the same instruction sets for both error free and error prone testing provided a comprehensive understanding of the performance of a given MIPS architecture in every tested operational mode. Hamilton [20] also determined the energy consumed by each error prone architecture using separately developed power equations similar to those presented in Section 2.6.5.4. These equations are not included in this discussion, because energy calculations/measurements were determined to be out of scope for this research as will be discussed in Ch. 3 of this document.

HITL testing of the various error prone architectures could only provide timing data for TMR and TSR MIPS when both the processor and memory were implemented on a single FPGA board. HITL timing data for AHR MIPS was not obtained because the architecture was too large to be implemented on a single board due to FPGA resource constraints. No HITL energy measurements were obtained for any of the error prone architectures due to the difficulties implementing the architecture across two FPGA boards, as was discussed in Section 2.6.4.3. The focus of the research presented in this document is to successfully implement error prone TMR, TSR, and AHR MIPS in hardware by implementing a serial communication scheme to overcome the HITL challenges faced by Hamilton in [20].

2.6.6 Error Prone Simulation/Calculation Results

The results collected for the simulation of each error prone architecture provided an understanding on the performance benefits of AHR MIPS over implementing only TMR or TSR redundancy. AHR MIPS can experience the most error scenarios, and because of this was found to have a rich performance trade space dependent upon the TMR to TSR transition point and the point at which an error was detected. Hamilton's results in [20] found that the performance of AHR MIPS due to a TMR Type A Early, TMR Type B-Best Early, TSR Best, and the error free performance of AHR MIPS were similar because those error scenarios have a minimal impact on the runtime performance of AHR MIPS. The AHR MIPS TMR Type A Late and TMR Type B-Best late error scenarios were found to use more energy and run faster than error free AHR MIPS, because both these error scenarios extend the time AHR MIPS stays in the TMR operating mode. The AHR MIPS TMR Type B-Worst Early, TMR Type B-Worst Late, and TSR Worst error scenarios were found to use more energy and take more time than error free AHR MIPS because these errors required the most program re-computation.

Hamilton also demonstrated the performance flexibility of AHR MIPS by showing how increasing the TMR to TSR point makes AHR MIPS perform more like TMR MIPS and reducing the TMR to TSR point makes AHR MIPS perform more like TSR MIPS. The results demonstrated that AHR provided distinct performance advantages over using just TMR or TSR redundancy. AHR MIPS was found to always use less energy than TMR MIPS and process programs faster than TSR MIPS, however AHR MIPS could not process as fast as TMR MIPS or with the same energy efficiency as TSR MIPS. Overall, the performance and energy tradeoffs of AHR MIPS compared to TMR and TSR MIPS were found to be acceptable given how flexible the performance of AHR MIPS can be.

The error prone simulation results were able to prove the viability of AHR as a redundancy scheme and were able to provide an answer to all research questions posed by Hamilton. However, HITL testing was unable to provide the necessary results to experimentally verify the performance seen in simulation. HITL testing was performed using one program for TMR MIPS and TSR MIPS, where each was tested in error free operating conditions and in every possible error scenario for that architecture. The HITL results found that the runtime performance for most of the error scenarios for TMR and TSR MIPS matched what was calculated/simulated. Any minor discrepancies found between the real-world and simulated performance were due to clock jitter and skew. However, for some of the error scenarios the discrepancy between real-world and simulated was too large to be due to the performance of the clock. The larger discrepancies were hypothesized to be due to timing delays introduced by the Quartus design software during the place-and-route step used to program the FPGAs, or due to the way in which the timing of error injection in software simulation was translated to HITL [20].

The HITL results provided some insight into how AHR MIPS could be experimentally proven in hardware, and along with the design of AHR itself, were used to both act as the foundation of this research and provide the direction/goals of this research.

III. Methodology

3.1 Introduction

This chapter presents the methods employed to modify the AHR MIPS architecture discussed in Ch. II Section 2.6 for implementation on a low Size, Weight, Power, and Cost (SWAP-C) Consumer Off the Shelf (COTS) FPGA. Section 3.2 discusses the FPGA that was chosen as the hardware target for this research. Section 3.3 describes how the original AHR MIPS architecture created in [20] was modified to enable hardware implementation. These modifications include the addition of four new components and changes to pre-existing components to create a new AHR MIPS architecture that operates correctly in hardware. Section 3.4 discusses how the new AHR MIPS design was implemented on the selected FPGA. This section explains how programs were created to run on the AHR MIPS architecture. This section also explains how program instructions and architecture specific processes were timed using simulations to support the development of the equations used to calculate program runtimes. Section 3.5 presents the final modifications made to the AHR MIPS architecture to facilitate hardware measurements, explains which MIPS architectures and operational modes were measured in hardware for comparison against calculations, defines the equations used to calculate the hardware runtimes of programs processed on each MIPS architecture and operational mode, and the methodology used to collect hardware measurements.

3.2 FPGA Hardware Selection

This section describes the process used to select the low SWAP-C COTS FPGA used in this research. The FPGA hardware chosen for this research was selected to overcome the hardware implementation issues encountered in the previous AHR

MIPS research [19, 20]. An explanation of those challenges is given in Section 3.2.1. Section 3.2.2 explains which FPGA was selected for use in this research, and provides a brief overview of the device specifications. Section 3.2.3 explains the new constraints that were encountered due to the FPGA hardware that was selected.

3.2.1 Previous Challenges

In the work preceding this research [20], TMR, TSR, and AHR MIPS were all successfully shown to operate in a simulated environment. However, after implementing the TMR and TSR MIPS architectures onto a single Terasic SoCKit Development board [52], a discrepancy was found between the simulated and measured energy usage of the MIPS processor for each architecture. The cause of this discrepancy was postulated to be due to the method used to measure the power consumed by the processor. By combining both the processor and memory onto one FPGA device, there was no reasonable way to determine how much energy was being consumed solely by the processor in hardware. The collected measurements represented the combined energy consumption of both the processor and memory. To collect the appropriate energy results, an additional FPGA board was used to physically separate the processor from memory. The decision to use two FPGA devices was carried over to this research to avoid any possible issues collecting component specific data.

The decision to separate the processor and memory using two FPGA boards required that all the signals output by the processor needed to be communicated off-board to the memory board and vice-versa. The method chosen to perform inter-board communication was to assign signals to GPIO pins on Terasic DE10-Standard development boards [25]. In total 100 separate wires were used to connect the two boards and perform data transmission between the processor and memory. Each MIPS processor type was tested using this communication scheme, but none of the

architectures operated correctly in hardware. After troubleshooting, it was determined that the Quartus II software used to program the FPGA may have been making hardware implementation decisions that caused real-world timing issues, despite simulations showing that timing for these architectures was being met. In conclusion, it was determined that an approach that utilizes a serial communication scheme between the boards may yield better results in hardware than an approach using parallel communication. The timing issues found in the work preceding this research informed the design decisions made to develop new hardware buffers/communication modules. These modules implement a serial communication scheme between the selected MIPS processor and memory where each resides on a separate FPGA board.

3.2.2 Xilinx Spartan - 7 Development Board

Having determined that it was necessary to use two FPGA boards to test any of the MIPS architectures in hardware, the next design decision was to select an FPGA board to use. This research used Xilinx Spartan 7 development boards because the boards/chips are low SWAP-C components. Also, Xilinx products are commonly used in the government space community and the design software/tools for Xilinx products are well supported. The specific Xilinx product used was the SP701 Spartan 7 Evaluation Board [61].

3.2.3 New Constraints

A potential benefit of separating the MIPS processor and memory across two FPGA boards was that it would be possible to collect power measurements and calculate the energy usage of the processor while executing programs. However, the SP701 Spartan 7 board used for this research is an evaluation/development board with many on-board peripherals and components. These peripherals and additional

components are all attached to the Spartan 7 FPGA chip and draw power while the board is powered on. Since all these components require power, it was determined that accurate power measurements of only the MIPS processor instantiated on the FPGA chip would be difficult to collect amongst all the other power demands on the board. Also, the SP701 FPGA board is powered by a 6-pin PCIe style power connector. This style of connector does not facilitate easy power measurements using an oscilloscope, whereas the previous research using the DE10 had a single DC 12V connection which was simpler to use for power measurements.

As a consequence of the new constraints incurred by use of the SP701 Development boards, power measurements of the MIPS processors during operation were not included as part of this research effort.

3.3 SP701 AHR MIPS Design

This section describes the new AHR MIPS architecture that was implemented on the two Xilinx Spartan 7 FPGAs. The new AHR MIPS architecture was designed to physically separate the AHR MIPS processor from memory. This was done for two purposes:

1. Previous attempts found that the AHR architecture and memory were too large to fit onto one low-cost COTS FPGA

2. Physical separation of the AHR MIPS processor and memory will support future radiation test campaigns of the processor

To ensure the AHR MIPS processor and memory properly functioned while existing on two physically separate FPGAs, required the design of additional modules to appropriately buffer and transmit data signals between the two major components. Four additional modules were designed to support inter-board communication. These

modules are the:

1. MIPS Buffer Out (MIBO)
2. Memory Buffer In (MEBI)
3. Memory Buffer Out (MEBO)
4. MIPS Buffer In (MIBI)

The AHR MIPS architecture designed and implemented in this research consists of the AHR MIPS processor, MIBO, MEBI, Memory, MEBO, and MIBI modules. The purpose, design, and standard operation of the four new modules are described in Sections 3.3.2 & 3.3.4. Only the purpose and standard operation of the AHR MIPS processor and memory module are described, because these components were previously developed in [20] and were not heavily modified as part of this research.

3.3.1 AHR MIPS Processor

The AHR MIPS processor module decodes instructions retrieved from memory and performs all arithmetic or logical operations required to process that instruction. The AHR MIPS processor is a radiation tolerant component that can operate in either a TMR or TSR mode depending on the current radiation environment.

The AHR MIPS processor inputs are a clock signal, reset signal, ready signal, 32-bit data signal, and PROGRAM DONE signal. The outputs are a read signal, write signal, 32-bit address, and 32-bit data signal. A diagram of the AHR MIPS processor is shown below in Figure 4. The clock and reset signals are not shown.

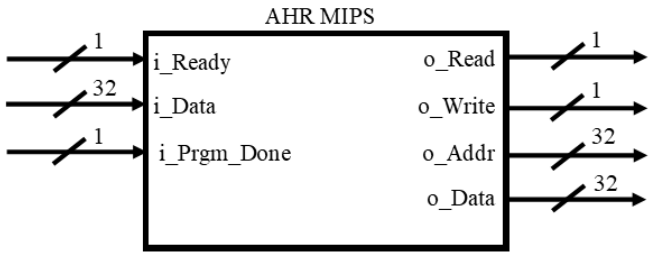

Figure 4. Diagram of the AHR MIPS Processor

After the AHR MIPS processor decodes an instruction, the output signals shown in Figure 4 are transmitted to the memory module to complete the specified read or write operation in memory. The AHR MIPS processor processes AHR program instructions supplied by memory until the processor sees the PROGRAM DONE input signal go high. When the PROGRAM DONE signal is high, the AHR MIPS processor resets itself and retrieves the first instruction from memory to restart the program. A more in-depth background on the design of the AHR MIPS processor was given in Ch. II Section 2.6.

3.3.2 Forward Path

Data output from the MIPS processor is transmitted to memory through a communications channel referred to as the Forward Path. The Forward Path consists of two modules. The first module is the MIPS Buffer Out (MIBO) which accepts all output signals from the AHR MIPS processor and serializes the data to send to Memory. The design of MIBO is described in Section 3.3.2.1. The second module is the Memory Buffer In (MEBI), which receives the serialized data from MIBO,

converts the data back to non-serialized AHR MIPS outputs, and supplies them to memory. The design of MEBI is described in Section 3.3.2.2. The communications path between the MIBO and MEBI modules implements the Universal Asynchronous Receiver-Transmitter (UART) hardware communication protocol. Contained in the MIBO and MEBI modules are a UART transmitter (TX) and UART receiver (RX). These UART modules are used in tandem with the Forward Path components to implement a radiation tolerant communications path that connects the AHR MIPS processor and memory

3.3.2.1 MIPS Buffer Out (MIBO)

The MIBO module is responsible for transmitting the output of the AHR MIPS processor to the MEBI module. The output signals of the AHR MIPS processor are the read and write operation flags, the address from which data needs to be read or to which data needs to be stored, and data to be written to memory when performing a write operation. The relevant output signals are transmitted using a UART TX module to the MEBI. The MIBO is implemented as an FSM which executes either a read or write process dependent upon the AHR MIPS processor read and write operation flag outputs. The MIBO module includes the logic necessary to determine whether an error occurred during transmission to MEBI. If the read flag output from the AHR processor is high, then the MIBO FSM enters state s_read0 and iterates through the states listed in Table 10 in Appendix A. The MIBO FSM read process completes four steps:

1. Transmit a unique 8-bit read code to MEBI (assumed to be error-free as described in Ch. I Section 1.3).

2. Transmit the 16-bit address from which data will be read in memory to MEBI.

3. Receive the 16-bit address back from MEBI.

4. Determine whether a radiation strike caused an error during transmission of the read address.

    (a) Compare the read address that was sent to MEBI against the read address received back from MEBI.

    (b) If they match, send a unique 8-bit correct code to MEBI and then return to the idle state of the FSM.

    (c) If they do not match, send a unique 8-bit error code to MEBI and repeat Steps 2-4 until it is determined that the read address was not corrupted during transmission.

If the write flag output from the AHR processor is high, then the MIBO FSM enters state s_write0 and iterates through the states listed in Table 11 in Appendix A. The MIBO FSM write process completes seven steps, where the first four are nearly identical to the read process:

1. Transmit a unique 8-bit write code to MEBI (assumed to be error-free as described in Ch. I Section 1.3).

2. Transmit the 16-bit address to which data will be written in memory to MEBI.

3. Receive the 16-bit address back from MEBI.

4. Determine whether a radiation strike caused an error during transmission of the write address.

    (a) Compare the write address that was sent to MEBI against the write address received back from MEBI.

    (b) If they match, send a unique 8-bit correct code to MEBI and proceed to Step 5.

(c) If they do not match, send a unique 8-bit error code to MEBI and repeat Steps 2-4 until it is determined that the write address was not corrupted during transmission.

5. Transmit the 32-bit data signal which will be written to the previously specified address in memory to MEBI.

6. Receive the 32-bit data signal back from MEBI.

7. Determine whether a radiation strike caused an error during transmission of the write data.

   (a) Compare the write data that was sent to MEBI against the write data received back from MEBI.

   (b) If they match, send a unique 8-bit correct code to MEBI and then return to the idle state of the FSM.

   (c) If they do not match, send a unique 8-bit error code to MEBI and repeat Steps 5-7 until it is determined that the write address was not corrupted during transmission.

The MIBO module also accepts a PROGRAM DONE signal that initiates a return to the FSM idle state, while the AHR MIPS processor re-starts program execution from the beginning.

3.3.2.2 Memory Buffer In (MEBI)

The MEBI module is responsible for receiving the output from MIBO, organizing the received data, and interfacing with memory. The output of MEBI to memory consists of a read and write enable, a 32-bit address, and a 32-bit data signal. The MEBI is an FSM which executes either a read or write operation in coordination

with MIBO. MEBI does not include any logic to determine whether a transmission error occurred because all received addresses and data are sent back to MIBO where error detection occurs. Operation of the MEBI FSM is dependent upon the unique 8-bit operation code sent by MIBO. While in the idle state, if the code received is the read code, then the FSM enters state s_read0 and iterates through the states listed in Table 12 in Appendix A. The MEBI FSM read process completes four steps:

1. Receive 16-bit read address from MIBO.
2. Transmit the 16-bit read address back to MIBO for error checking.
3. Receive the unique 8-bit code from MIBO which indicates whether an error occurred.
   (a) If the code received matches the correct code, then set the memory read enable signal high and supply the read address.
   (b) If the code received matches the error code, then repeat Steps 1-3 until it is determined that the read address was sent correctly.

While in the idle state, if the code received is the write code, then the FSM enters state s_write0 and iterates through the states listed in Table 13 in Appendix A. The MEBI FSM write process completes six steps, where the first three are nearly identical to the read process:

1. Receive 16-bit write address from MIBO.
2. Transmit the 16-bit write address back to MIBO for error checking.
3. Receive the unique 8-bit code from MIBO which indicates whether an error occurred.
   (a) If the code received matches the correct code, then proceed to Step 4.

(b) If the code received matches the error code, then repeat Steps 1-3 until it is determined that the write address was sent correctly.

4. Receive 32-bit write data from MIBO.

5. Transmit the 32-bit write data back to MIBO for error checking.

6. Receive the unique 8-bit code from MIBO which indicates whether an error occurred.

   (a) If the code received matches the correct code, then set the memory write enable signal high and supply the write address and data. Return FSM to idle state.

   (b) If the code received matches the error code, then repeat Steps 4-6 until it is determined that the write data was sent correctly.

After completion of either a read or write operation, MEBI sets all output signals to memory back to low/zeros to prevent the MIPS architecture from processing the same instruction again.

3.3.3 Memory

The memory module stores the program instructions that are executed by the AHR MIPS processor, acts as external storage for register data should a program require more space, and stores the SRPs created during program execution along with the SRP location pointer. An AHR program loaded into memory contains space for TMR instructions, equivalent TSR instructions, temporary and permanent memory, TSR MIPS SRP creation instructions, TSR MIPS error recovery instructions, and SRP memory. The AHR program length determines the size of the memory module implemented in the FPGA fabric. For this research, AHR programs and hardware

design files were randomly generated using a series of MATLAB scripts and used a 33 - instruction subset of the standard MIPS Instruction Set Architecture (ISA). The instruction subset used for this research is listed in Table 8 of Appendix A.

During normal operation, an AHR MIPS processor will only read from the sections of memory that contain instructions, temporary, permanent, or SRP memory. Also, AHR MIPS will only write to temporary, permanent, and SRP memory. The memory module determines when a program has completed execution by setting the output PROGRAM DONE signal high when a read request is made at a memory location outside the bounds of the AHR program. The previously described Forward Path presents data to the memory module. This data indicates whether a read or write operation is required while also supplying the corresponding read address or write address and data. The memory module accesses the specified address and either outputs the data to complete a read operation or writes the supplied data to the specified address to complete a write operation. The memory module does not output any data during a write operation. The hardware implementation of the AHR memory module does not utilize any radiation hardening techniques, and is instead assumed to be immune to radiation as stated in Ch. I Section 1.3. The memory module outputs the data to the Return Path, which will transmit the data back to AHR MIPS to complete that instruction or begin processing the next instruction.

3.3.4 Return Path

The Return Path is the communications channel used to send data from memory to the AHR MIPS processor. The Return Path consists of two modules. The first module is the Memory Buffer Out (MEBO). The inputs to MEBO are the memory outputs and the output of MEBO is a serialized version of the memory outputs. The design of MEBO is described in Section 3.3.4.1. The second module is the MIPS

Buffer In (MIBI). The input to MIBI is a serialized version of the memory outputs from MEBO. MIBI then converts the serialized data to non-serialized memory outputs and supplies them to the AHR MIPS processor. The design of MIBI is described in Section 3.3.4.2.

3.3.4.1 Memory Buffer Out (MEBO)

The MEBO module is responsible for transmitting the output from memory to the MIBI module. The output signals of memory are the ready signal, 32-bit data output signal, and PROGRAM DONE signal. A diagram of memory is shown below in Figure 5.

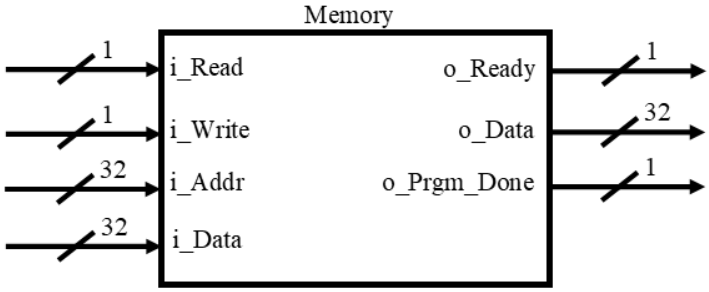

Figure 5. Diagram of the AHR MIPS Memory

The MEBO is an FSM which executes a read, write, or PROGRAM DONE process dependent upon whether MEBI set the read or write enable high, or if the PROGRAM DONE signal coming out from memory is high. The MEBO module includes the logic necessary to determine whether an error occurred during transmission to MIBI. If the read enable output from MEBI was high, then the MEBO FSM enters state s_read0 and iterates through the states listed in Table 14 in Appendix A. The MEBO FSM read process completes 4 steps:

1. Transmit a unique 8-bit read code to MIBI (assumed to be error-free as de-

scribed in Ch. I Section 1.3).

2. Transmit the 32-bit data read from the specified address supplied to memory, to MIBI.

3. Receive the 32-bit read data back from MIBI.

4. Determine whether a radiation strike caused an error during transmission of the read data.

   (a) Compare the read data that was sent to MIBI against the read data received back from MIBI.

   (b) If they match, send a unique 8-bit correct code to MIBI and then return to the idle state of the FSM.

   (c) If they do not match, send a unique 8-bit error code to MIBI and repeat Steps 2-4 until it is determined that the read data was not corrupted during transmission.

If the write enable output from MEBI is high, then the MEBO FSM enters state s_write0 and iterates through the states listed in Table 15 in Appendix A. The MEBO FSM write process completes only one step:

1. Transmit a unique 8-bit ready code to MIBI (assumed to be error-free as described in Ch. I Section 1.3).

The MEBO FSM write process is only one step because a write operation does not return any data from memory in the AHR MIPS architecture. Therefore, the only information that needs to be transmitted to the MIPS processor is that the write operation was completed.

If the PROGRAM DONE signal output from memory is high, then the MEBO FSM enters state s_reset0 and iterates through the states listed in Table 16 in Appendix A. The MEBO FSM reset/PROGRAM DONE process completes only one step:

1. Transmit a unique 8-bit PROGRAM DONE code to MIBI (assumed to be error-free as described in Ch. I Section 1.3).

The MEBO FSM reset/PROGRAM DONE process transmits the PROGRAM DONE code to MIBI, which MIBI uses to reset both the MIPS processor and MIBO. Meanwhile, MEBO also signals MEBI to reset, which ensures that all signals to memory are cleared, and no further instructions are executed until the MIPS processor makes a new request.

3.3.4.2 MIPS Buffer In (MIBI)

The MIBI module is responsible for receiving the output from MEBO, organizing the received data, and interfacing with the AHR MIPS processor. The MIBI outputs to the AHR MIPS processor are a ready signal, 32-bit data signal, and a PROGRAM DONE signal. The MIBI is an FSM which executes a read, write, or reset/PROGRAM DONE operation in coordination with MEBO. MIBI does not include any logic to determine whether a transmission error occurred because all received data is sent back to MEBO where error detection occurs. The MIBI FSM operation is dependent upon the unique 8-bit operation code sent by MEBO. While in the idle state, if the code received is the read code, then the FSM enters state s_read0 and iterates through the states listed in Table 17 in Appendix A. The MIBI FSM read process completes three steps:

1. Receive 32-bit read data from MEBO.

2. Transmit the 32-bit read data back to MEBO for error checking.

3. Receive the unique 8-bit code from MEBO which determines whether an error occurred.

   (a) If the code received matches the correct code, then set the ready signal to the processor high and supply the read data. Return FSM to idle state.

   (b) If the code received matches the error code, then repeat Steps 1-3 until it is determined that the read data was sent correctly.

While in the idle state, if the code received is the write code, then the FSM enters state s_write0 and iterates through the states listed in Table 18 in Appendix A. The MIBI FSM write process completes only one step:

1. After receipt of the unique 8-bit ready code, set the output ready signal high.

The MIBI FSM write process only needs to signal to the processor that a write operation was completed in memory.

While in the idle state, if the code received is the reset/PROGRAM DONE code, then the FSM enters state s_reset0 and iterates through the states listed in Table 19 in Appendix A. The MIBI FSM reset process completes only one step:

1. After receipt of the unique 8-bit reset code, set the output PROGRAM DONE signal high to restart AHR MIPS operation.

The MIBI FSM reset/PROGRAM DONE process only needs to initiate an AHR MIPS processor and MIBO reset to restart program execution.

3.3.4.3 Summary of Communication Module Design

The communication modules were all designed to provide an error/radiation tolerant method of inter-board data transmission between the AHR MIPS processor

and the corresponding memory module. The communication modules adhere to the UART communication protocol, and thus can be easily modified to operate at a user-defined baud rate. The baud rate is set by specifying the number of clock-cycle counts that will occur between the transmission of bits. The number of counts is set low to allow the new AHR MIPS architecture to be simulated. The number of counts is set high to produce a baud rate that can be physically realized in FPGA hardware. The AHR MIPS architecture cannot be physically implemented without the communication modules for the reasons specified in Section 3.2.1. In conjunction with the design of the communication modules, components of the AHR MIPS architecture also needed to be modified to produce a hardware design that had stable and repeatable performance. The AHR MIPS architecture required these performance characteristics to facilitate hardware measurements and data collection.

3.3.5 Modifications to Previous AHR MIPS Architecture

The AHR MIPS architecture was designed to begin operation in TMR mode upon start-up, and transition to TSR mode after processing a user-defined number of program instructions without error. The design established in [20] has the AHR MIPS system remain in TSR mode even after the current program being executed has completed. This design decision was made because in a real-world scenario the only time the AHR MIPS system would return to TMR mode is either after multiple radiation-induced errors are detected, or the increased processing performance in TMR mode is required to meet mission requirements. However, in this research the AHR MIPS architecture needed to be implemented in hardware, and its operation processing an AHR instruction set needed to be repeatable to verify that the hardware implementation functioned in the same way as was simulated. The runtime of an AHR instruction set is measured assuming that the AHR MIPS system begins in

TMR mode and transitions to TSR mode during that time. This means that for hardware verification, the AHR MIPS processor needed to be modified to completely reset after an AHR program was fully executed. This change forced the AHR MIPS processor to repeat the same behavior and program runtimes until the device was powered off. To produce this repeatable behavior, the AHR controller component of the AHR MIPS processor was modified as described in Section 3.3.5.1.

An error inject module is used to simulate the effects of up-to two independent simultaneous radiation-induced upsets, and although this module was previously created it also required modifications to produce expected behavior in hardware. The design of the error inject module was described in Ch. II Section 2.6.5.1. The error injector is programmed by the user to modify a specific bit within a specific register of the AHR MIPS datapath at a specific PC value. However, the error injector cannot inject an error if the AHR MIPS system is performing TMR SRP creation or TMR error recovery and should not inject an error while executing a transition between operating modes. This design decision was made for two reasons. First, the portion of a program's overall runtime dedicated to TMR SRP creation, TMR error recovery, or transitions between operating modes is small. Thus, in a real-world scenario the likelihood of a radiation strike producing an error in a user register during those operations is small in comparison to the likelihood of an error occurring during the processing of program instructions. Secondly, the error injector module would have required significant modifications to exactly simulate an upset during TMR SRP creation, TMR error recovery, or transitions between operating modes. To simulate an upset occurring during those processes using the current error inject module only requires that an error be injected into the target register before that process begins. These errors are not difficult to inject, and thus any modifications to the error injector to exactly simulate those error scenarios would have unnecessarily increased the

design complexity of the error injector module. This research maintained the design decision for the error inject module and did not modify it to inject errors during those special processes. However, the error injector was modified to operate correctly during another error scenario.

Through preliminary analysis, it was determined that there was an edge case where the error injector would prematurely inject an error while trying to generate a specific error scenario. Section 3.3.5.2 describes how the error injector was modified to exhibit proper behavior for that edge case.

3.3.5.1 Modifications to the AHR Controller

The AHR controller is a component of the AHR MIPS architecture. The AHR controller is implemented as an FSM that monitors the operation of the AHR MIPS system and determines whether to operate in TMR or TSR mode and when to transition from one mode to the other. The operation of the controller is described in detail in Ch. II Section 2.6.1.3 & 2.6.3. The controller monitors various control signals output by other MIPS components to determine what the current system state should be. Two of the main control signals pertinent to the new design are the RESET and PROGRAM DONE signals. In the previous AHR MIPS design, the controller would only fully reset the entire system back to TMR mode upon receipt of the global RESET signal. To produce stable and repeatable behavior, the AHR controller in the new design was modified so that the PROGRAM DONE signal would trigger system behavior equivalent to a global RESET.

3.3.5.2 Modifications to the Error Injector

The error injector is a component of the Basic MIPS processors used within the AHR MIPS architecture. The error injector uses an ERROR OVERRIDE control

signal to determine when an error can be injected depending on whether the program is actively being executed, or if a special process is occurring. The error injector only modifies the data held in a user register at a specific PC value and if the ERROR OVERRIDE control signal is low. The ERROR OVERRIDE control signal is externally generated and supplied by the TMR Voter. While AHR MIPS is operating in TMR mode, the TMR Voter will set the ERROR OVERRIDE signal high when it determines that a special process needs to be executed. One of these processes is the creation of a TMR SRP.

A SRP is created in TMR mode after a user-defined number of instructions are processed. Since the TMR Voter is implemented as a FSM, it takes a couple clock cycles for the ERROR OVERRIDE signal to be set high after the requisite number of instructions have been processed. Meanwhile, the error injector is only waiting for the specified PC value to occur, and if the ERROR OVERRIDE signal is low, then the error is injected. This means that an error will be injected prematurely if a TMR SRP needs to be created on the same PC value at which an error needs to be injected. The error injector will identify that the current PC value matches the user-specified value and will inject an error a few clock cycles before the TMR Voter has determined a SRP needs to be made and raises the ERROR OVERRIDE signal. In this edge case the error should be injected at that PC value, but only after a SRP has been created and processing of program instructions resumes. To exhibit expected behavior in this edge case, the error injector was modified to have a few clock cycles of internal delay before determining if an error needs to be injected. The internal delay is long enough to allow for the ERROR OVERRIDE signal to be raised and received by the error injector, should the system need to perform TMR SRP creation or TMR error recovery. These modifications to the error injector were required to realize specific error scenarios for the AHR MIPS architecture during hardware testing.

3.3.5.3 Summary of AHR MIPS Architecture Modifications

The AHR controller and error injector modules were modified to produce repeatable behavior in hardware and to realize specific error scenarios required for system verification. The original AHR controller is the component design that should be used on-orbit as part of a space system. However, the previous controller design required the modifications described in Section 3.3.5.1 to gather hardware data on the average runtime of AHR instruction sets. The original error injector component design did not produce the expected behavior in the edge case described in Section 3.3.5.2, and thus required modification to realize that scenario in hardware.

3.3.6 Finalized Design of AHR MIPS Architecture

Figure 6 below shows the final design of the AHR MIPS architecture which was implemented in hardware using two Xilinx Spartan 7 SP701 - Development Boards. This design incorporates the original AHR MIPS design, the four communication modules, and the revisions made to the AHR controller and error injector.

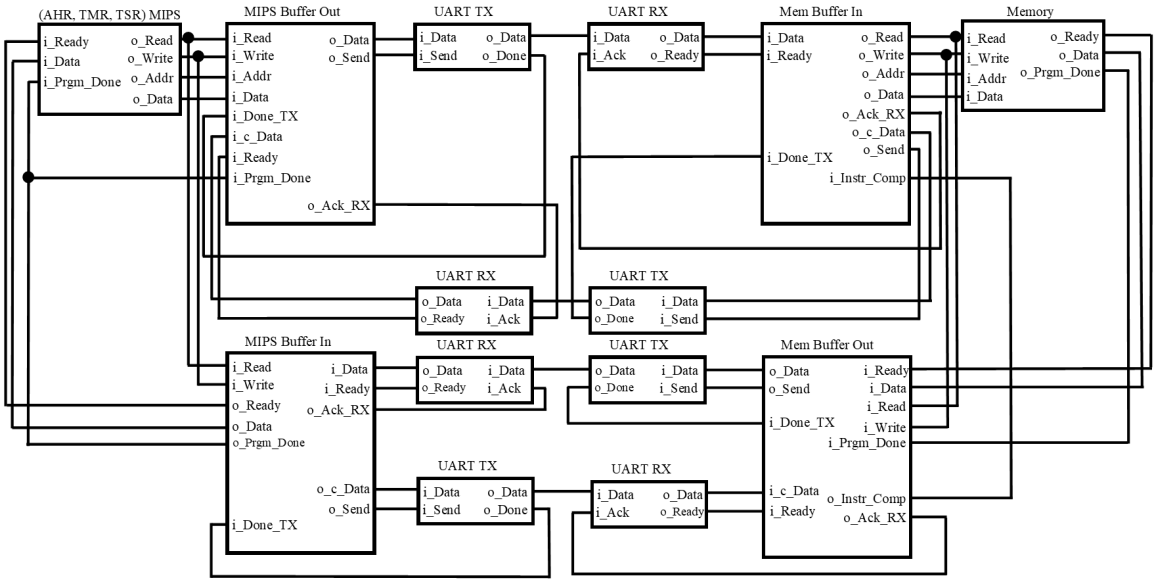

Figure 6. AHR MIPS Hardware Design

3.4 Implementation of AHR MIPS on Xilinx FPGAs

Hardware verification of the SP701 AHR MIPS hardware design described in Section 3.3 requires the following:

1. AHR MIPS can be implemented in hardware and operates as designed and verified through simulations.

2. Runtime performance of AHR MIPS in hardware is measurable and matches a calculated prediction.

The runtime performance of the original AHR MIPS design was calculated using measurements of the time it took to process MIPS instructions and architecture

specific processes. This research collected the timing measurements of all necessary operations using Xilinx Vivado simulations. This research applied the same methodology used by Hamilton in [20] to measure the MIPS instructions and architecture specific processes. The methodology employed by Hamilton was described in Ch. II Sections 2.6.4.2 and 2.6.5.4. Section 3.4.2 provides detailed information on what operations were measured, how they were measured, and how equations were made to predict the runtime of those operations at different communication baud rates. These operation timing equations were required to create the program runtime equations that were used to predict the hardware program runtimes.

Additional modifications were made to the AHR MIPS architecture to correctly measure program runtimes using an oscilloscope. Section 3.5 describes the modifications that were made to AHR MIPS, as well as the methodology used to experimentally measure the runtimes of programs executed on TMR, TSR, and AHR MIPS under various operational conditions.

3.4.1 Instruction Set/Program Generation

The architecture specific programs (TMR, TSR, and AHR) used for simulation and hardware testing were generated in the same way as discussed in Ch. II Section 2.6.2. The same MATLAB scripts and code used in [20] were used to generate programs that remain compatible with the various MIPS architectures. The instruction sets were randomly generated while adhering to the following constraints:

1. Min & Max Number of Program Instructions - 100/150
2. Number of Program Loops - 3
3. TSR SRP Creation Points - [-1, -2, -3]
4. Number of Instructions before Generating TMR SRPs - 40

5. TMR to TSR Transition Instruction Count - 60

This research focuses on verifying that the AHR MIPS architecture can function in hardware, that the real-world program runtimes can be accurately calculated, and that the overall behaviors of each MIPS architecture match what was found in the previous research [20]. For this reason, the number of program loops was set to 3 to allow for PROGRAM DONE signals to be generated in reasonable time in both simulation and hardware. The previous AHR MIPS architecture was designed with the AHR MIPS processor directly connected to memory. In the new design the communication modules were required to facilitate inter-board communication between the processor and memory. These communication modules increased the runtime of a program significantly, and therefore all instruction set parameters were reduced to make verification of the architecture simpler.

100 instruction sets were randomly generated using the constraints previously listed and 20 were randomly selected for functional testing in hardware. A subset of the 20 sets were selected for functional verification using Vivado simulations. The subset included the shortest program, longest program, and two programs with lengths in-between those two extremes. Each instruction set in the subset was simulated for an entire program run and the simulated runtimes were compared against the predicted/calculated runtimes produced by the program runtime equations. This subset was selected with the intent to exercise the extremes of the timing equations in MATLAB and create edge case scenarios that could be used to either correct or refine those timing equations. The selected instruction subset was used in this way to build confidence that all the other instruction set runtimes across all MIPS architectures would be correctly calculated.

3.4.2 AHR MIPS Timing Calculations

The timing information required to calculate the runtime of an AHR instruction set is listed below:

1. TMR Mode
    (a) TMR Instruction Timing
    (b) TMR Non-Instruction Process Timing
2. TSR Mode
    (a) TSR Instruction Timing
    (b) TSR Non-Instruction Process Timing
3. AHR Non-Instruction Process Timing

The timing of a TMR or TSR instruction refers to the measured simulation time it took for the corresponding MIPS system to process each of the 33 individual MIPS instructions used in this research. The instruction timings were obtained by measuring how long it took for a Basic MIPS controller FSM to progress from its base state (state_0), process the instruction, and return to state_0. Each MIPS architecture uses at least one Basic MIPS processor, and thus has at least one Basic MIPS controller to use for timing measurements. All MIPS processors were simulated/configured to operate at 50 MHz.

The timing of a TMR, TSR, or AHR non-instruction process refers to the timing of architecture specific processes such as creating a TMR SRP, performing TSR error recovery, and AHR MIPS transitioning from TMR to TSR mode. These are just a few examples with a full list of the non-instruction processes listed in Table 9 in Appendix A. The measurement points used to collect the timing for each of these

non-instruction processes varied depending on whether execution of that process was controlled by a set of instructions or a component.

For example, TSR SRP creation is executed by processing a set of instructions explicitly listed in a TSR program. Therefore, TSR SRP creation timing was measured from when the Basic MIPS controller was in state_0 for the first TSR SRP instruction, until state_0 of the first instruction following execution of all the TSR SRP creation instructions. An example of a non-instruction process that is dictated by a component is TMR MIPS Type B Error Recovery. If a Type B error is detected, the TMR Voter FSM will enter the corresponding error state (FSM_ERR1), will perform the appropriate error recovery, and will return to its base state. Timing of TMR MIPS Type B Error Recovery starts when the TMR Voter is in state FSM_ERR1, and ends at state_0 of the Basic MIPS controller at the first program instruction following the successful reload of all three Basic MIPS processors.

The new AHR MIPS architecture developed in this research had timing measurements for instructions and non-instruction processes that were much different than what was measured previously by Hamilton in [20]. For brevity, instructions and non-instruction processes will be referred to collectively as "operations" for the remainder of this section. The difference in timing measurements was caused by the communication modules implemented in the new AHR MIPS architecture. Each of the communication modules used a UART TX and RX component and because the UART protocol is used, the new timings measured were much higher than what was previously seen. The timings collected for the new AHR MIPS architecture also scale with the baud rate at which the UART modules operate. The baud rate of each UART module was set by specifying the number of clock cycles, or counts that pass between UART bit-to-bit transmissions. For all simulations this count was set to a simulation rate of 10, where the MIPS processors were simulated to operate at 50

MHz. A simulation rate of 10 was selected to allow the various MIPS architectures to simulate within reasonable time.

The goal of collecting the timing for all operations was to build timing equations that could be used to calculate the runtime of any program on any MIPS architecture. Since the timing of operations are affected by the UART baud rate, it was necessary to further inspect the timing of each process and mathematically define which portions were affected by a baud rate change. The portion of the overall time of an operation that scales with the baud rate is referred to as the variable delay. The portion of the overall time of an operation that does not scale with the baud rate is referred to as the fixed delay.

The general formula used to determine the timing of any operation is given by Equation 9, where T_C is the total time of an operation while the communication modules are operating at count C, F_{DT} represents the total fixed delay of that operation, and V_{DTC} represents the total variable delay of that operation at count C.

$$T_C = F_{DT} + V_{DTC} \tag{9}$$

At the simulation rate where $C = 10$, F_{DT} was found by subtracting the measured V_{DTC} from T_C. The V_{DTC} was obtained by first measuring the time it took for a single byte of information to be sent from one UART TX module to the corresponding UART RX module, this process is hereafter referred to as a UART transaction. The time it took to complete a single UART transaction was then multiplied by the total number of UART transactions that occurred during a given operation to determine the total V_{DTC}. Once V_{DTC} was found F_{DT} could be calculated. An illustrative example of this process to measure the delays and determine the timing equation for that process is given below.

Example: A TSR MIPS architecture was simulated and processed a set of in-

structions, while operating with $C = 10$. To determine the timing equation for a basic instruction, the total time was measured using the Basic MIPS controller, as described earlier. T_C for a basic instruction at $C = 10$ was found to be:

$$T_{10} = 33.02\mu s \tag{10}$$

The simulation was inspected and 16 UART transactions occurred to process that instruction. The time it took for one UART transaction (U_{Tran}) was found to be:

$$U_{Tran} = 1.9\mu s \tag{11}$$

Equation 9 can now be used to determine F_{DT} using T_C, U_{Tran}, and the number of UART transactions that occurred.

$$\begin{aligned} T_C &= F_{DT} + V_{DTC} \\ (33.02\mu s) &= F_{DT} + [(16)(1.9\mu s)] \\ F_{DT} &= 33.02\mu s - [(16)(1.9\mu s)] \\ F_{DT} &= 33.02\mu s - 30.4\mu s \\ F_{DT} &= 2.62\mu s \end{aligned} \tag{12}$$

In this example, $F_{DT} = 2.62\mu s$ and $V_{DTC} = 30.4\mu s$ as shown in Equation 12.

The process described in the example was used to determine F_{DT} and V_{DTC} for all operations at the simulation rate $C = 10$. Once all fixed and variable delays were calculated, Equation 13 was created to calculate/predict the timing of any operation at higher C values.

$$T_C = F_{DT} + (\tfrac{C}{10})V_{D10} \tag{13}$$

Equation 13 expands the V_{DTC} term of Equation 9 to define the variable delay

contributed by processing an operation at count C. Since the simulation count value is $C = 10$, all calculated variable delay values must be modified with respect to that simulation value. V_{D10} represents the measured variable delay of that operation at $C = 10$. This simulation variable delay, V_{D10}, is then modified by $\frac{C}{10}$ to return the total variable delay of an operation at a new C value. The denominator of the modifier is 10, because the simulation UART count value is $C = 10$. This modifier was intended to easily calculate V_{DTC} and thus T_C at higher C values, but it was discovered that the timing values calculated using Equation 13 differed from simulation at higher C values. Equation 13 was used to calculate operation timings at $C = 12$ and $C = 14$, and these calculations did not match what was simulated at those C values. Using those calculated timings, the error growth as C increased was found and indicated that an additional variable delay term was necessary to account for variable delay being added to the system at higher C that was not taken into account by Equation 13.

Equation 14 was created to correctly calculate the T_C of any operation at any C value, and includes a third term that accounts for the additional variable delay not included in the second term of Equation 13. The additional variable delay term was determined to be dependent upon the number of UART transactions that an operation performed. Specifically, it was found that for every 16 UART transactions an additional $(\frac{C-10}{10})$ of delay was added. Basic instructions and Branch instructions each perform 16 UART transactions, while LW/SW and all non-instruction processes perform 32 or more UART transactions. The $(\frac{n}{16})$ modifier in the third term required that the number of UART transactions, n, for every operation be counted.

$$T_C = F_{DT} + [(\tfrac{C}{10})V_{D10} + (\tfrac{n}{16})(\tfrac{C-10}{10})] \tag{14}$$

Equation 14 was used to calculate T_C for operations at higher C values and was

found to match simulations of those processes.

In total, timing information was collected and used to create timing equations for all 33 instructions in the reduced-MIPS set used in this research and for 10 architecture specific non-instruction processes.

3.5 Hardware Testing and Data Collection

Hardware testing was performed to both experimentally verify that the AHR MIPS architecture was operating as designed, and to demonstrate that the program runtimes matched what was calculated using timing equations in MATLAB. Demonstrating that the program runtimes in hardware matched calculations would show that the performance of the new AHR MIPS architecture is predictable in hardware. The runtime of any program executed by the AHR MIPS architecture in hardware was measured from the rising edge of a PROGRAM DONE signal until the rising edge of the next PROGRAM DONE signal. In this research the PROGRAM DONE signal was measured at the input of the AHR MIPS processor. Despite measuring off the AHR MIPS processor, the PROGRAM DONE signal is an output of the memory module. Memory keeps the PROGRAM DONE signal low when MIPS accesses memory locations within the bounds of the program and sets the signal high when MIPS attempts to access a memory location outside the program bounds. The PROGRAM DONE signal was easily inspected using the Vivado simulation environment, but to measure the PROGRAM DONE signal input to the MIPS processor in hardware required additional modifications to the AHR MIPS hardware design.

3.5.1 Modifications to AHR MIPS for Hardware Measurements

The PROGRAM DONE signal is transmitted from memory to AHR MIPS through the Return Path of the communication network described in Section 3.3.4. To fa-

cilitate hardware measurements, an additional DONE signal trace was added that branched off from the PROGRAM DONE signal output by MIBI to the MIPS processor. This DONE signal trace was routed to one of the PMOD pins available on the SP701 Spartan 7 development board. Since two Spartan 7 boards are used, the board that contained the processor also contained the MIBI module to ensure that the DONE signal was measured at the same time as it was presented to the MIPS processor. However, before reaching a PMOD pin the DONE signal needed to be modified to produce a signal that was stable and easily measurable by an oscilloscope.

In the AHR MIPS architecture, the PROGRAM DONE signal is only set high for a few clock cycles before returning to a low value. The amount of time a PROGRAM DONE signal is set high is marginal compared to the rest of the program runtime. The oscilloscope used in this research can measure with high-fidelity at a small time scale, but loses fidelity as the time scale of measurement increases. This fidelity scaling occurs because the oscilloscope is limited by the amount of RAM it can use to store measured data. This limitation required that a modified DONE signal be created that could be measured, but did not modify the behavior of the internally generated PROGRAM DONE signal. A modified DONE signal that could be measured was created using a new hardware component called the LED oscillator. The LED oscillator accepts the DONE signal as input and outputs a modified version of the DONE signal to both a PMOD pin and LED on the SP701 development board. The LED oscillator acts as a simple toggle whose output switches between high and low when a rising edge of the incoming DONE signal is detected. The output of the LED oscillator is a signal that can be more easily detected and measured by the oscilloscope to determine program runtimes. The AHR MIPS hardware design could be physically implemented after making the previously described modifications, thus enabling collection of runtime data.

3.5.2 Architectures & Operational Modes Calculated and Measured

Verification of the new AHR MIPS architecture required that the calculated program runtimes for all versions of the MIPS architecture, in each unique operational mode, be shown to match the times measured in hardware. Table 5 shows all the MIPS architectures and their respective operational modes whose program runtimes were measured in hardware.

Table 5. Architectures and Operational Modes Measured in Hardware

MIPS Architecture	**Operational Mode**
TMR	No Error (NE)
	Error A
	Error B - Best
	Error B - Worst
TSR	No Error (NE)
	Best Single Error
	Worst Single Error
AHR	No Error (NE)
	TMR Mode - Error A - Early
	TMR Mode - Error A - Late
	TMR Mode - Error B - Best Early
	TMR Mode - Error B - Best Late
	TMR Mode - Error B - Worst Early
	TMR Mode - Error B - Worst Late
	TSR Mode - Best Single Error
	TSR Mode - Worst Single Error

Each operational mode shown in Table 5 was caused by injecting zero, one, or two errors using the error injector discussed in Section 3.3.5.2.

3.5.2.1 TMR Operational Modes/Error Scenarios

The error free operation (EF) of TMR MIPS was expected to produce the fastest overall program runtimes. The timing equations used to calculate the error free runtime of a given TMR program were Equations 1, 2, and 3, presented in Ch. II Section 2.6.4.2.

The first meaningful operational mode, or error scenario, of the basic TMR MIPS architecture occurred when a single error was injected during program execution. A single error is referred to as a Type A error. A Type A error results in a minimal impact to the program runtime. This is because a single error in one of the three Basic MIPS processors can be detected and corrected by the TMR Voter. The TMR Voter will select the correct result produced by the other two processors and use those processors to overwrite and correct the processor that produced an error. Figure 7 below generalizes the runtime of a TMR program from start to end, and shows that a Type A error can be corrected with minimal impact to the runtime of a program, where the red "X" indicates that an error was detected in Basic MIPS processor 1 and the green circles indicate that MIPS 0 & 2 were in agreement when the error was detected. The blue arrows indicate the error recovery operation that occurred and provides insight into how that error recovery process added to the program runtime, where in the case of a Type A error, the runtime was not noticeably affected. This error was produced in hardware across all selected instruction sets, by injecting a single error at any point during program execution.

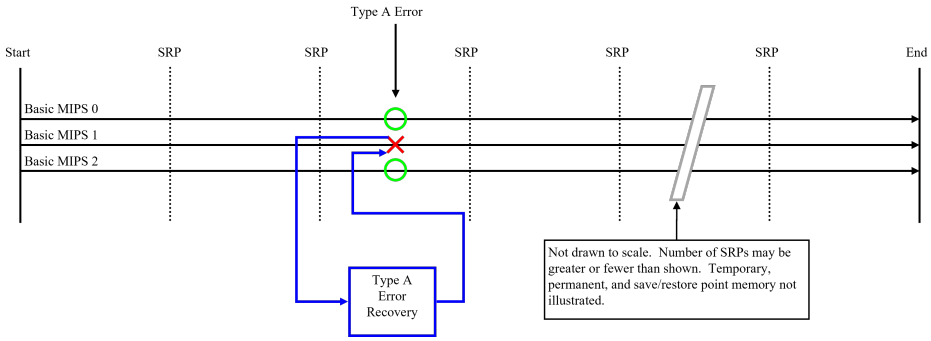

Figure 7. TMR MIPS Type A Error Scenario [20]

The equation used to calculate the runtime of any TMR MIPS program that experienced a Type A error is given by Equation 15 [20], where $T_{TMR\ MIPS}$ is the error free runtime of the TMR program that is calculated using Equation 1, $T_{TMR\ ttdA}$ is the time it takes the TMR Voter to detect that a single error has occurred, $T_{TMR\ recA}$ is the time it takes to perform TMR Type A error recovery, and $T_{TMR\ retA}$ is the time it takes the architecture to resume program execution after completing the Type A error recovery process. The last three terms were measured in simulation and general timing equations were made for them as described in Section 3.4.2.

$$T_{TMR\ ErrA} = T_{TMR\ MIPS} + T_{TMR\ ttdA} + T_{TMR\ recA} + T_{TMR\ retA} \tag{15}$$

The remaining error scenarios tested for the TMR MIPS architecture occurred when two simultaneous errors were injected into separate processors during program execution. This type of error is referred to as a Type B error. A Type B error results in a larger impact on program runtime and the magnitude of that impact depends on when the errors are detected. Recovery from a Type B error always requires that the processors revert to a previous state held by a SRP. The best-case Type B error occurs soon after a SRP has been created, because only a few instructions will need to be re-processed after reverting to that SRP. The worst-case Type B error occurs

near the end of SRP creation and creates a scenario where the maximum number of instructions need to be re-processed. The worst-case Type B error requires that the oldest/inactive SRP be loaded, and that all instructions up-to creation of the SRP where the error was detected be re-processed.

Figure 8 below shows these two Type B error scenarios and how they can affect program runtime.

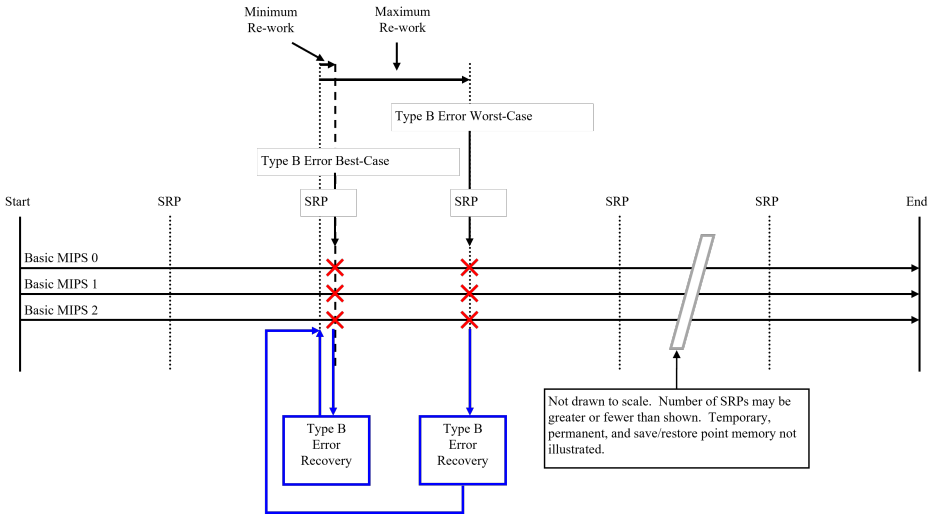

Figure 8. TMR MIPS Type B Error Scenarios [20]

The equation used to calculate the runtime of any TMR MIPS program that experienced a Type B Best error is given by Equation 16 [20], where $T_{TMR\ ttdB}$ is the time it takes the TMR Voter to detect that a Type B error has occurred, $T_{TMR\ recB}$ is the time it takes to perform TMR Type B error recovery, and $T_{TMR\ retB\ Best}$ is the time it takes the architecture to resume program execution after completing the Type B error recovery process. $T_{TMR\ ttdB}$ and $T_{TMR\ recB}$ were measured in simulation and general timing equations were made for them as described in Section 3.4.2.

$$T_{TMR\ ErrBBest} = T_{TMR\ MIPS} + T_{TMR\ ttdB} + T_{TMR\ recB} + T_{TMR\ retB\ Best} \tag{16}$$

$T_{TMR\ retB\ Best}$ in Equation 16 is calculated using the following process:

1. Locate every point in the TMR program at which a SRP is created, called a Save Index (SI)
2. Calculate the time between a SI and the nearest SW that comes after that SI
3. Repeat Step 2 for all SI's in the program
4. Select the minimum time between SI and SW as $T_{TMR\ retB\ Best}$

The equations used to calculate $T_{TMR\ retB\ Best}$ were developed by Hamilton in [20] and were not modified for this research. Equation 39 in Ch. 5 Section 5.4.1.1 in [20] was specifically used to calculate $T_{TMR\ retB\ Best}$.

The equation used to calculate the runtime of any TMR MIPS program that experienced a Type B Worst error is given by Equation 17 [20], where $T_{TMR\ SRP\ Err}$ is the time it takes the TMR Voter to detect an error while writing the last register of the MIPS processors to memory during the TMR SRP creation process, and $T_{TMR\ retB\ Worst}$ is the time it takes the architecture to resume program execution after completing the Type B error recovery process. $T_{TMR\ SRP\ Err}$ was measured in simulation and the general timing equation for it was made as described in Section 3.4.2.

$$T_{TMR\ ErrBWorst} = T_{TMR\ MIPS} + T_{TMR\ SRP\ Err} + T_{TMR\ recB} + T_{TMR\ retB\ Worst} \tag{17}$$

$T_{TMR\ retB\ Worst}$ in Equation 17 is calculated using the following process:

1. Calculate the time between one SI and the next SI
2. Repeat Step 1 until all times between consecutive SI's have been calculated

3. Select the maximum time between consecutive SI's as $T_{TMR\ retB\ Worst}$

The equations used to calculate $T_{TMR\ retB\ Worst}$ were developed by Hamilton in [20] and were not modified for this research. Equation 43 in Ch. 5 Section 5.4.1.1 in [20] was specifically used to calculate $T_{TMR\ retB\ Worst}$.

The MATLAB code modified/used to calculate program runtimes also returned both the loop count and instruction at which a TMR Type B error should be detected to realize both the best and worst-cases in hardware.

It was predicted that the runtime speeds of all TMR operational modes would be from fastest to slowest: EF TMR, TMR Type A error, TMR Type B Best error, and TMR Type B Worst error. Relative to the other architectures and operational modes, EF TMR MIPS and all TMR MIPS error scenarios were expected to process programs the fastest because TMR MIPS only processes TMR programs, whereas AHR MIPS must process a portion of the program in TSR mode using TSR instructions and TSR MIPS must process the entire program using TSR instructions.

3.5.2.2 TSR Operational Modes/Error Scenarios

The error free operation of TSR MIPS was expected to produce one of the slowest overall program runtimes. The timing equations used to calculate the error free runtime of a given TSR program were Equations 4 & 5, presented in Ch. II Section 2.6.4.2.

The TSR MIPS architecture only uses SRPs to correct any detected errors during program runtime and cannot mask a single error like the TMR MIPS architecture can. Any error detected in a TSR MIPS architecture will increase the program runtime.

The first error scenario tested for the TSR MIPS architecture was the Best-case single error. The Best-case TSR error is an error that minimizes the amount of instruction re-processing that must be performed after loading a SRP. Thus, the

Best-case error is injected before and detected by the first BNE-SW pair of a TSR program, either after program start or after a TSR SRP has been made. Since EDDI-TSR is specifically implemented for standalone TSR MIPS and AHR MIPS, errors are detected in register pairs using a BNE instruction processed before executing a SW to write the value in the register pair to memory.

The Worst-case TSR error occurs while writing the last user register to memory during SRP creation. Since TSR SRP creation occurs at the end of a TSR loop, this Worst-case error scenario guarantees that one or more program loops must be re-processed, thus drastically increasing the program runtime. Both the Best and Worst-case TSR error scenarios are depicted in Figure 9 below.

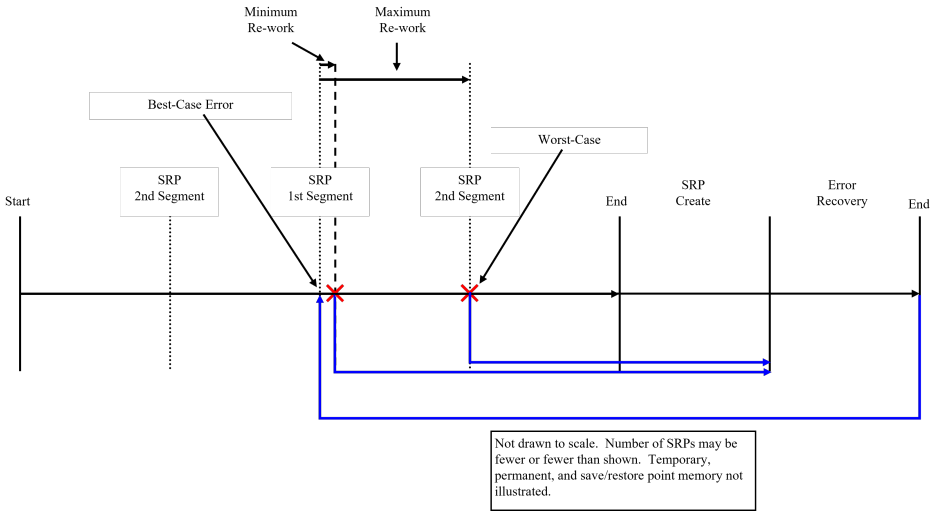

Figure 9. TSR MIPS Error Scenarios [20]

The instruction sets used for hardware testing in this research only performed three program loops, therefore the Worst-case TSR error was realized in hardware across all sets by injecting an error before creation of SRP1. The number of instructions in a TSR program to create SRP1 is slightly more than the number of instructions to create SRP0, thus SRP1 always takes longer to create than SRP0. The Best-case error was produced in hardware for each instruction set by injecting an error before

the first BNE-SW pair.

The equation used to calculate the runtime of any TSR MIPS program that experienced a Best-case error is given by Equation 18, where $T_{TSR\ MIPS}$ is the error free runtime of the TSR program that is calculated using Equation 4, $T_{TSR\ Rec}$ is the time it takes TSR error recovery to complete, $T_{TSR\ ret}$ is the time it takes to re-process instructions from the loaded SRP up-to the point at which the error was detected, $T_{TSR\ SRP0}$ is the time it takes to create SRP0, $T_{TSR\ SRP1}$ is the time it takes to create SRP1, and $T_{TSR\ skip}$ is the time it takes to process the six instructions that are used to determine when the three SRPs are made. $T_{TSR\ Rec}$, $T_{TSR\ SRP0}$, $T_{TSR\ SRP1}$, and $T_{TSR\ skip}$ were measured in simulation and general timing equations were made for them as described in Section 3.4.2.

$$\begin{aligned} T_{TSR\ Best} &= T_{TSR\ MIPS} + T_{TSR\ Rec} + T_{TSR\ ret} + T_{TSR\ SRP0} + \cdots \\ &2 \cdot T_{TSR\ SRP1} - T_{TSR\ skip} \end{aligned} \tag{18}$$

$T_{TSR\ ret}$ in Equation 18 is calculated using Equation 19 [20], where N_{TSR} is the total number of instructions in one TSR program loop, $t_{I_{TSR\ n}}$ is the amount of time it takes to process instruction number n of the program, $n_{TSR\ init}$ is the number of instructions that are used to initialize a TSR program (4 total), and SW_{TSR} is an array that contains the index of all SW instructions in the TSR program.

$$T_{TSR\ ret} = \sum_{n=N_{TSR}-3}^{N_{TSR}} t_{I_{TSR\ n}} + \sum_{n=n_{TSR\ init}+1}^{SW_{TSR}(1)} t_{I_{TSR\ n}} \tag{19}$$

The equation used to calculate the runtime of any TSR MIPS program that experienced a Worst-case error is given by Equation 20, where $T_{TSR\ Loop}$ is the time required to complete a single TSR MIPS program loop as previously given by Equation 2. $T_{TSR\ SRP1\ Err}$ is the time it takes to perform creation of SRP1 up-to the last point at which an error can be detected, which is while writing the last user registers

to memory. For the short programs used in this research, the Worst-case error is always realized by detecting an error during the first program loop while creating SRP1. Although SRP1 is created during the first and last program loops, the first loop processes more instructions because none of the instructions used to determine if a SRP should be made are skipped. Injecting an error during the first loop while creating SRP1 increases the total program length by a full $T_{TSR\ Loop}$. $T_{TSR\ SRP1\ Err}$ was measured in simulation and a general timing equation was made for it as described in Section 3.4.2.

$$\begin{aligned} T_{TSR\ Worst} &= T_{TSR\ MIPS} + T_{TSR\ Rec} + T_{TSR\ Loop} + \cdots \\ &T_{TSR\ SRP1\ Err} + T_{TSR\ SRP0} + 2 \cdot T_{TSR\ SRP1} - T_{TSR\ skip} \end{aligned} \tag{20}$$

It was predicted that the runtime speeds of all TSR operational modes would be from fastest to slowest: EF TSR, TSR Best-case error, and TSR Worst-case error. Relative to the other architectures and operational modes, EF TSR MIPS and both error scenarios of TSR MIPS were expected to process programs the slowest because TSR MIPS only processes TSR programs, whereas AHR MIPS processes a portion of the program in TMR mode using TMR instructions and TMR MIPS processes the entire program using TMR instructions.

3.5.2.3 AHR Operational Modes/Error Scenarios

In this research, the AHR MIPS architecture always processed an AHR instruction set in TMR mode first, transitioned to TSR mode if possible, and completed the program in TSR mode. None of the AHR MIPS error scenarios listed earlier in Table 5 result in AHR MIPS transitioning from TSR mode back to TMR mode, and thus none of the timing calculations take that process into consideration. The tested AHR MIPS error scenarios were established by Hamilton in [20] and this research is focused

on showing that every error scenario previously simulated can be realized in hardware, thus no new error scenarios were created and tested.

The error free operation of AHR MIPS was expected to have program runtimes that fell between TSR MIPS and TMR MIPS. The timing equations used to calculate the error free runtime of a given AHR program are given in Equation 21, which is a modified version of Equation 8. In Equation 21, n_{ASRP} is the number of TMR SRPs made during the AHR program, $t_{TMR \rightarrow TSR}$ is the time it takes AHR MIPS to complete the transition from TMR to TSR mode, P_{loops} is the number of program loops that complete before transitioning, $T_{TSR\ BrI}$ is the time it takes to process a single TSR branch instruction, n_{SRP} is the number of TMR SRPs created if AHR does not transition to TSR mode, and $T_{TMR\ BrI}$ is the time it takes to process a single TMR branch instruction.

P_{loops} was calculated using Equation 6 and n_{ASRP} was calculated using Equation 7 where $n_{ASRP} = n_{CSRP}$. Both $t_{TMR \rightarrow TSR}$ and $T_{TSR\ BrI}$ were measured in simulation and general timing equations were made for these operations as described in Section 3.4.2. The value of n_{SRP} was calculated using Equation 3.

The instruction sets generated for this research only perform a total of three program loops for the reasons discussed in Section 3.4.1. As a result, most of the AHR timing equations that will be discussed use if statements to calculate the portion of time AHR MIPS spent in TSR mode. These if statements are dependent upon the calculated number of loops that complete before transitioning to TSR mode. The total program runtime of an EF AHR MIPS program ($T_{EF\ AHR\ MIPS}$) is calculated by adding the amount of time the architecture spent in TMR mode and TSR mode, represented by $t_{AHR\ TMR}$ and $t_{AHR\ TSR}$ respectively.

$$
\begin{aligned}
&t_{AHR\ TMR} = t_{TMR\ init} + P_{loops} \cdot T_{TMR\ loop} + \cdots \\
&T_{TMR\ SRP} \cdot n_{ASRP} + t_{TMR \rightarrow TSR} \\
&if\ P_{loops} = 0 \\
&\quad t_{AHR\ TSR} = (n_{loops} - P_{loops}) \cdot t_{TSR\ loop} + T_{TSR\ SRP0} + \cdots \\
&\quad 2 \cdot T_{TSR\ SRP1} + T_{TSR\ BrI} - T_{TSR\ skip} \\
&elseif\ P_{loops} = 1 \\
&\quad t_{AHR\ TSR} = (n_{loops} - P_{loops}) \cdot t_{TSR\ loop} + T_{TSR\ SRP0} + \cdots \\
&\quad T_{TSR\ SRP1} + T_{TSR\ BrI} - T_{TSR\ skip} \\
&elseif\ P_{loops} = 2 \\
&\quad t_{AHR\ TSR} = (n_{loops} - P_{loops}) \cdot t_{TSR\ loop} + T_{TSR\ SRP1} + \cdots \\
&\quad T_{TSR\ BrI} - \tfrac{2}{3} T_{TSR\ skip} \\
&else \\
&\quad t_{AHR\ TSR} = 0 \\
&\quad t_{AHR\ TMR} = t_{TMR\ init} + n_{loops} \cdot T_{TMR\ loop} + \cdots \\
&\quad T_{TMR\ SRP} \cdot (n_{SRP} - 1) + T_{TMR\ BrI} \\
&end \\
&T_{EF\ AHR\ MIPS} = t_{AHR\ TMR} + t_{AHR\ TSR}
\end{aligned}
\tag{21}
$$

There are multiple error scenarios that can occur while AHR MIPS operates in TMR mode. As seen in Table 5, the number of error scenarios possible for AHR MIPS while operating in TMR mode is greater than the number standalone TMR MIPS can experience. There are more error scenarios for AHR MIPS because TMR errors can either have a minor or significant effect on runtime performance depending on where they are detected. In AHR MIPS the TMR to TSR transition point is dependent upon the number of TMR instructions that are processed without error, so the location at which a TMR error is detected will determine how much of the

program is completed in TMR mode. For AHR MIPS, each TMR Type A and B error scenario can be expanded into error scenarios that are labeled as Early or Late. Early errors are the AHR TMR mode errors that are detected early and have a minimal effect on the TMR to TSR transition point. Late errors are the AHR TMR errors that are detected late and have a significant effect on the TMR to TSR transition point.

AHR TMR Mode Type A Errors: An AHR TMR mode Type A - Early error is one that is detected at the first SW of the program. An AHR TMR mode Type A - Late error is one that is detected at the last SW before the TMR to TSR transition point occurs. AHR TMR Type A errors are masked due to the use of a TMR voter and do not initiate a system reload from a SRP. AHR TMR Type A errors are quickly corrected, but cause the TMR to TSR transition point to occur later while executing an AHR program. Since each Type A error causes the TMR to TSR transition point to occur later, more of the program is performed in TMR mode. As a result, both Type A errors cause the runtime of an AHR MIPS program to be faster than if it was processed without error. Figures 10 & 11 below depict what each AHR Type A error variation looks like as applied to a generic AHR program.

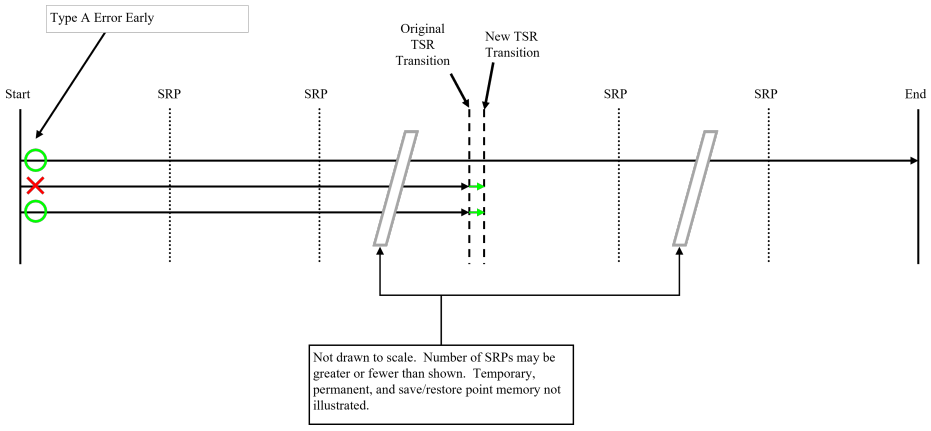

Figure 10. AHR MIPS TMR Type A Error - Early [20]

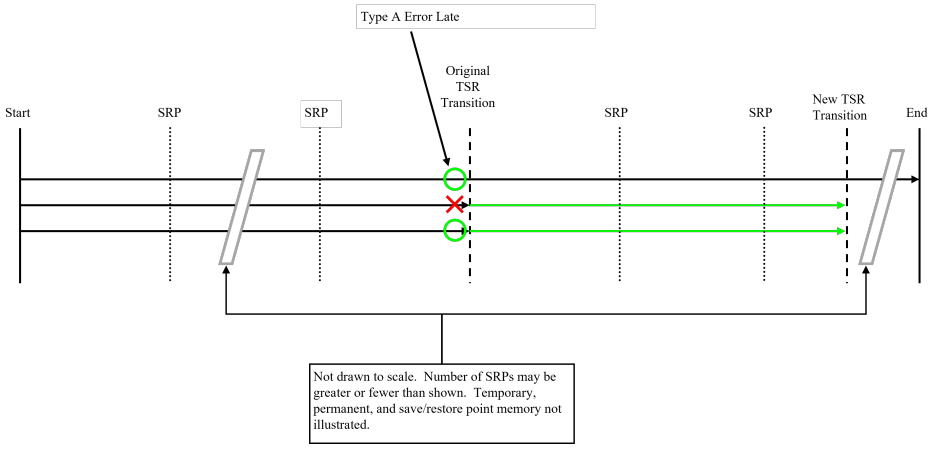

Figure 11. AHR MIPS TMR Type A Error - Late [20]

The process used to calculate the runtime of any AHR MIPS program that experienced a TMR Type A - Early error is given by Equation 22 [20], where $P_{loops\ TMR\ A\ Early}$ is the number of program loops that occur before transitioning from TMR to TSR caused by a Type A error as determined by Equation 23 [20], and $n_{ASRP\ A\ Early}$ is the number of TMR SRPs created by AHR MIPS before the transition occurs which is calculated using Equation 24 [20].

The if/else process presented in Equation 22 was written to appropriately calculate the runtime for any of the instruction sets generated for this research, by considering how the calculated $P_{loops\ TMR\ A\ Early}$ value affects the program loop distribution between the TMR and TSR modes. The variable $t_{nom\ AE}$ represents the nominal AHR - TMR mode operation time, or specifically represents the runtime of the TMR program that executes outside of the processes that correct the Type A error. The variable $t_{err\ AE}$ represents the time spent correcting the Type A error. The total runtime $T_{ATMR\ A\ Early}$ is calculated by adding the time AHR MIPS spent in both

TMR and TSR mode, $t_{ATMRAE\ TMR}$ and $t_{ATMRAE\ TSR}$ respectively.

$$
\begin{aligned}
&t_{nom\ AE} = t_{TMR\ init} + P_{loops\ TMR\ A\ Early} \cdot T_{TMR\ loop} + \cdots \\
&T_{TMR\ SRP} \cdot n_{ASRP\ A\ Early} \\
&t_{err\ AE} = T_{TMR\ ttdA} + T_{TMR\ recA} + T_{TMR\ retA} \\
&if\ P_{loops\ TMR\ A\ Early} = 0 \\
&\quad t_{ATMRAE\ TMR} = t_{nom\ AE} + t_{err\ AE} + t_{TMR \rightarrow TSR} \\
&\quad t_{ATMRAE\ TSR} = (n_{loops} - P_{loops\ TMR\ A\ Early}) \cdot T_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + 2 \cdot T_{TSR\ SRP1} + T_{TSR\ BrI} - T_{TSR\ skip} \\
&elseif\ P_{loops\ TMR\ A\ Early} = 1 \\
&\quad t_{ATMRAE\ TMR} = t_{nom\ AE} + t_{err\ AE} + t_{TMR \rightarrow TSR} \\
&\quad t_{ATMRAE\ TSR} = (n_{loops} - P_{loops\ TMR\ A\ Early}) \cdot T_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + T_{TSR\ SRP1} + T_{TSR\ BrI} - T_{TSR\ skip} \\
&elseif\ P_{loops\ TMR\ A\ Early} = 2 \\
&\quad t_{ATMRAE\ TMR} = t_{nom\ AE} + t_{err\ AE} + t_{TMR \rightarrow TSR} \\
&\quad t_{ATMRAE\ TSR} = (n_{loops} - P_{loops\ TMR\ A\ Early}) \cdot T_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP1} + T_{TSR\ BrI} - \tfrac{2}{3} T_{TSR\ skip} \\
&else \\
&\quad t_{ATMRAE\ TMR} = t_{TMR\ init} + n_{loops} \cdot T_{TMR\ loop} + \cdots \\
&\quad T_{TMR\ SRP} \cdot (n_{SRP} - 1) + t_{err\ AE} + T_{TMR\ BrI} \\
&\quad t_{ATMRAE\ TSR} = 0 \\
&end \\
&T_{ATMR\ A\ Early} = t_{ATMRAE\ TMR} + t_{ATMRAE\ TSR}
\end{aligned}
\tag{22}
$$

Equation 23 calculates $P_{loops\ TMR\ A\ Early}$ in nearly the same way as error free P_{loops} in Equation 6. The $P_{loops\ TMR\ A\ Early}$ calculation must consider that additional instructions equivalent to the number of instructions up-to the first SW of the program

($SW_{TMR}(1)$) must be processed before transitioning to TSR mode.

$$P_{loops\ TMR\ A\ Early} = \left\lceil \frac{SW_{TMR}(1)+n_{transition}-n_{TMR\ init}}{N_{TMR}} \right\rceil \tag{23}$$

Equation 24 uses the $P_{loops\ TMR\ A\ Early}$ value, but is calculated in the same manner as described for n_{CSRP} in Equation 7.

$$n_{ASRP\ A\ Early} = \left\lfloor \frac{P_{loops\ TMR\ A\ Early} \cdot N_{TMR}+n_{TMR\ init}}{n_{save}} \right\rfloor \tag{24}$$

The process used to calculate the runtime of any AHR MIPS program that experienced a TMR Type A - Late error is given by Equation 25 [20]. This process is nearly identical to what was presented in Equation 22, but differs in the way $P_{loops\ TMR\ A\ Late}$ is calculated using Equation 26 [20]. The new transition point $P_{loops\ TMR\ A\ Late}$ is found by considering that a TMR Type A - Late error is detected at the last SW of the TMR program ($SW_{TMR}(length(SW_{TMR}))$) before transitioning, whereas the Best error occurred when the error was detected at the first TMR SW. Equation 27 [20] calculates the number of TMR SRPs that occur before the new transition point caused by a late error. The total runtime $T_{ATMR\ A\ Late}$ is calculated by adding the time AHR MIPS spent in both TMR and TSR mode, $t_{ATMRAL\ TMR}$ and $t_{ATMRAL\ TSR}$ respectively.

$$
\begin{aligned}
&t_{nom\ AL} = t_{TMR\ init} + P_{loops\ TMR\ A\ Late} \cdot T_{TMR\ loop} + \cdots \\
&T_{TMR\ SRP} \cdot n_{CSRP\ A\ Late} \\
&t_{err\ AL} = T_{TMR\ ttdA} + T_{TMR\ recA} + T_{TMR\ retA} \\
&if\ P_{loops\ TMR\ A\ Late} = 0 \\
&\quad t_{ATMRAL\ TMR} = t_{nom\ AE} + t_{err\ AE} + t_{TMR \to TSR} \\
&\quad t_{ATMRAL\ TSR} = (n_{loops} - P_{loops\ TMR\ A\ Late}) \cdot T_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + 2 \cdot T_{TSR\ SRP1} + T_{TSR\ BrI} - T_{TSR\ skip} \\
&elseif\ P_{loops\ TMR\ A\ Late} = 1 \\
&\quad t_{ATMRAL\ TMR} = t_{nom\ AE} + t_{err\ AE} + t_{TMR \to TSR} \\
&\quad t_{ATMRAL\ TSR} = (n_{loops} - P_{loops\ TMR\ A\ Late}) \cdot T_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + T_{TSR\ SRP1} + T_{TSR\ BrI} - T_{TSR\ skip} \\
&elseif\ P_{loops\ TMR\ A\ Late} = 2 \\
&\quad t_{ATMRAL\ TMR} = t_{nom\ AE} + t_{err\ AE} + t_{TMR \to TSR} \\
&\quad t_{ATMRAL\ TSR} = (n_{loops} - P_{loops\ TMR\ A\ Late}) \cdot T_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP1} + T_{TSR\ BrI} - \tfrac{2}{3} T_{TSR\ skip} \\
&else \\
&\quad t_{ATMRAL\ TMR} = t_{TMR\ init} + n_{loops} \cdot T_{TMR\ loop} + \cdots \\
&\quad T_{TMR\ SRP} \cdot (n_{SRP} - 1) + t_{err\ AE} + T_{TMR\ BrI} \\
&\quad t_{ATMRAL\ TSR} = 0 \\
&end \\
&T_{ATMR\ A\ Late} = t_{ATMRAL\ TMR} + t_{ATMRAL\ TSR}
\end{aligned} \tag{25}
$$

$$
P_{loops\ TMR\ A\ Late} = \left\lceil \frac{SW_{TMR}(length(SW_{TMR})) + n_{transition} - n_{TMR\ init}}{N_{TMR}} \right\rceil \tag{26}
$$

$$
n_{CSRP\ A\ Late} = \left\lfloor \frac{P_{loops\ TMR\ A\ Early} \cdot N_{TMR} + n_{TMR\ init}}{n_{save}} \right\rfloor \tag{27}
$$

The AHR Type A - Early error was produced in hardware by injecting an error and detecting it at the first SW of each instruction set. The AHR Type A - Late error was produced in hardware using the MATLAB code to return both the loop count and instruction at which to inject an error, so that it was detected by the last SW in the TMR portion of the AHR program before the TMR to TSR transition occurred.

It was predicted that AHR MIPS experiencing an AHR Type A - Late error would process a program faster than AHR Type A - Early. An AHR Type A - Late error causes AHR MIPS to transition from TMR to TSR later in the program, thus more of the program is performed in TMR mode. An AHR Type A - Early error has a minimal effect on the TMR to TSR transition point and thus AHR MIPS performs more of the program in TSR mode compared to AHR Type A - Late. An AHR Type A - Early error produces a runtime that is slightly slower than EF AHR because if the error does not cause the architecture to perform an additional program loop in TMR mode, then the Type A - Early runtime takes longer than EF AHR by the same amount of time it takes to perform Type A error recovery. The predicted runtime speeds of the AHR MIPS Type A errors with respect to EF AHR were from fastest to slowest: AHR Type A - Late, EF AHR, and AHR Type A - Early.

If an AHR Type A - Early or Late error occurs and the program is short enough, AHR MIPS may not be able to transition from TMR to TSR mode. If AHR MIPS does not transition to TSR mode due to a Type A Early or Late error, then AHR MIPS processes the entire program in TMR mode using the TMR instructions. The corresponding Type A - Early & Late runtimes are not only equivalent to each other, but nearly equivalent to the runtime of TMR MIPS with a Type A error because each architecture would process the program using TMR instructions and perform the same error recovery. If the programs are too short to tolerate an AHR Type A error and AHR MIPS cannot transition to TSR mode, then the predicted runtime

speeds of the AHR MIPS Type A errors with respect to EF AHR were from fastest to slowest: AHR Type A - Early & Late and then EF AHR.

AHR TMR Mode Type B Best Errors: An AHR TMR mode Type B Best - Early error occurs when the errors are detected at the first SW in the TMR portion of an AHR program. An AHR Type B Best - Late error occurs when the errors are detected at the first SW that follows the creation of a TMR SRP before the original TMR to TSR transition point. When multiple TMR SRPs are created in an AHR program, the Best - Late error location is the one which adds the least time to the overall AHR program runtime. The AHR Type B Best - Early error scenario causes the runtime of the program to increase when compared to the error free scenario, but does not change the location of the TMR to TSR point. The AHR Type B Best - Late error scenario causes the TMR to TSR point to occur later and causes more of the program to be processed in TMR mode, thus producing runtimes faster than the error free scenario. Figures 12 & 13 below depict what each AHR Type B Best error variation looks like.

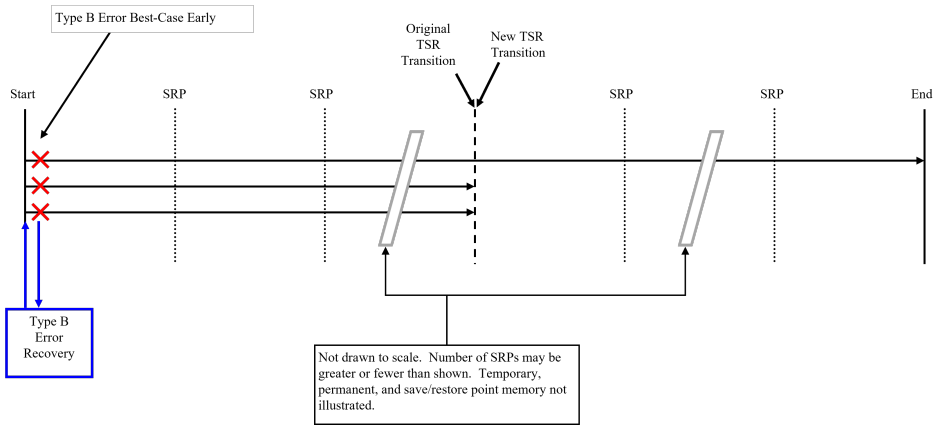

Figure 12. AHR MIPS TMR Type B Error Best - Early [20]

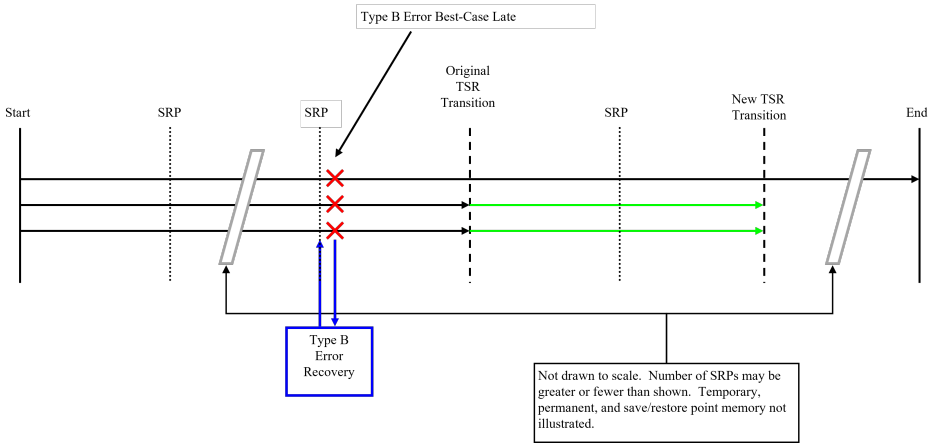

Figure 13. AHR MIPS TMR Type B Error Best - Late [20]

The process used to calculate the runtime of any AHR MIPS program that experienced a TMR Type B Best - Early error is given by Equations 28, 29, and 30 [20]. Equation 29 and Equation 30 should be considered as one continuous process, but were split into two in order to better fit this document.

Equation 28 is used to determine the new TMR to TSR transition point ($P_{loops\ TMR\ B\ Best}$) due to a Type B error detected at a SW for the n^{th} save index ($SI_{ATMR}(n)$) in TMR mode. The variable SI_{ATMR} is an array that contains the instruction indices where TMR SRPs are created before the original TMR to TSR transition point was expected to occur [20]. The variable SL_{ATMR} is an array that contains the program loop count values where the TMR SRPs are created before transitioning to TSR mode. The variable b_2 is an array containing the candidate SW indices that should be used as error injection points to realize either the slowest or fastest program completion times. SI_{ATMR} and SL_{ATMR} were calculated in Ch. 5 Section 5.4.1.3 of Hamilton's dissertation using Equation 53 [20]. b_2 was calculated in Ch. 5 Section 5.4.1.3 of Hamilton's dissertation using Equations 54, 55, and 56 [20].

Equation 28 is used to determine $n_{ASRP\ B\ Best}(n)$, which is the new number of TMR SRPs created due to a detected error with respect to the n^{th} save index [20].

Equation 28 is also used to determine the time it takes AHR MIPS to return to the SW at which the error was detected after recovering from the Type B error for the n^{th} save index, represented by the variable $T_{add}(n)$.

$$
\begin{aligned}
&for\ n = 1\ to\ length(b_2) \\
&\quad Flag = 0 \\
&\quad if\ SI_{ATMR} = 1 \\
&\quad\quad P_{loops\ TMR\ B\ Best}(n) = P_{loops} \\
&\quad else \\
&\quad\quad P_{loops\ TMR\ B\ Best}(n) = \cdots \\
&\quad\quad\quad \left\lceil \frac{SI_{ATMR}(n) + SL_{ATMR} \cdot N_{TMR} + n_{transition} - n_{TMR\ init}}{N_{TMR}} \right\rceil \\
&\quad end \\
&\quad n_{ASRP\ B\ Best}(n) = \left\lfloor \frac{P_{loops\ TMR\ B\ Best}(n) \cdot N_{TMR} + n_{TMR\ init}}{n_{save}} \right\rfloor \\
&\quad if\ SI_{ATMR}(n) \leq SW_{ATMR}(b2(n)) - 1 \\
&\quad\quad T_{add}(n) = \sum_{m=SI_{ATMR}(n)}^{SW_{ATMR}(b2(n))-1} t_{I_{TMR\ m}} \\
&\quad elseif\ SI_{ATMR}(n) = SW_{ATMR}(b2(n)) \\
&\quad\quad T_{add}(n) = 0 \\
&\quad elseif\ SL_{ATMR}(n) < P_{loops\ TMR\ B\ Best}(n) \\
&\quad\quad T_{add}(n) = \sum_{m=SI_{ATMR}(n)}^{n_{TMR\ init}+N_{TMR}} t_{I_{TMR\ m}} + \sum_{m=n_{TMR\ init}+1}^{SW_{ATMR}(b2(n))-1} t_{I_{TMR\ m}} \\
&\quad else \\
&\quad\quad T_{add}(n) = 0 \\
&\quad\quad Flag = 1 \\
&\quad end \\
&end
\end{aligned}
\tag{28}
$$

Equations 29 and 30 use the values $P_{loops\ TMR\ B\ Best}$, $n_{ASRP\ B\ Best}$, and T_{add} for ev-

ery save index n, to calculate an array of AHR MIPS Type B Best-case error runtimes ($T_{ATMR\ B\ Best\ Times}$). The $Flag$ is used to indicate whether the current save index and SW are valid, where $Flag = 1$ if the SW following the save index occurs after the new $P_{loops\ TMR\ B\ Best}$, and the resulting calculations should not be considered valid. $T_{ATMR\ B\ Best\ Times}$ is calculated by adding $t_{ATMRBB\ TMR}(n)$ and $t_{ATMRBB\ TSR}(n)$, which are the times AHR MIPS takes to complete the TMR and TSR portions of the program respectively for the n^{th} save index. The array $T_{ATMR\ B\ Best\ Viable\ Times}$ is a modified version of $T_{ATMR\ B\ Best\ Times}$ used to eliminate any possibility of selecting a Type B error runtime produced by detecting the error after the TMR to TSR transition P_{loops}. The AHR MIPS Type B Best - Early error runtime is found by selecting the maximum calculated $T_{ATMR\ B\ Best\ Times}$. The AHR MIPS Type B Best - Worst error runtime is found by selecting the minimum calculated $T_{ATMR\ B\ Best\ Times}$. The indices b_3 and b_4 were used to determine the instruction index and loop count at which a Type B error should be injected to realize the Best - Early and Best - Worst runtimes in hardware respectively.

$$
\begin{aligned}
&for\ n = 1\ to\ length(b_2)\\
&\quad if\ Flag = 1\\
&\qquad t_{ATMRBB\ TMR}(n) = NaN\\
&\qquad t_{ATMRBB\ TSR}(n) = NaN\\
&\quad else\\
&\qquad t_{nom\ BB} = t_{TMR\ init} + P_{loops\ TMR\ B\ Best}(n) \cdot T_{TMR\ loop} + \cdots\\
&\qquad T_{TMR\ SRP} \cdot n_{ASRP\ B\ Best}(n)\\
&\qquad t_{err\ BB} = T_{TMR\ ttdB} + T_{TMR\ recB} + T_{add}(n)\\
&\qquad if\ P_{loops\ TMR\ B\ Best}(n) = 0\\
&\qquad\quad t_{ATMRBB\ TMR}(n) = t_{nom\ BB} + t_{err\ BB} + t_{TMR \to TSR}\\
&\qquad\quad t_{ATMRBB\ TSR}(n) = (n_{loops} - P_{loops\ TMR\ B\ Best}(n)) \cdot t_{TSR\ loop} + \cdots\\
&\qquad\quad T_{TSR\ SRP0} + 2 \cdot T_{TSR\ SRP1} + T_{TSR\ BrI} - T_{TSR\ skip}\\
&\qquad elseif\ P_{loops\ TMR\ B\ Best}(n) = 1\\
&\qquad\quad t_{ATMRBB\ TMR}(n) = t_{nom\ BB} + t_{err\ BB} + t_{TMR \to TSR}\\
&\qquad\quad t_{ATMRBB\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Best}(n)) \cdot t_{TSR\ loop} + \cdots\\
&\qquad\quad T_{TSR\ SRP0} + T_{TSR\ SRP1} + T_{TSR\ BrI} - T_{TSR\ skip}\\
&\qquad elseif\ P_{loops\ TMR\ B\ Best}(n) = 2\\
&\qquad\quad t_{ATMRBB\ TMR}(n) = t_{nom\ BB} + t_{err\ BB} + t_{TMR \to TSR}\\
&\qquad\quad t_{ATMRBB\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Best}(n)) \cdot t_{TSR\ loop} + \cdots\\
&\qquad\quad T_{TSR\ SRP1} + T_{TSR\ BrI} - \tfrac{2}{3} T_{TSR\ skip}\\
&\qquad else\\
&\qquad\quad t_{ATMRBB\ TMR}(n) = t_{TMR\ init} + n_{loops} \cdot T_{TMR\ loop} + \cdots\\
&\qquad\quad T_{TMR\ SRP} \cdot (n_{SRP} - 1) + t_{err\ BB} + T_{TMR\ BrI}\\
&\qquad\quad t_{ATMRBB\ TSR} = 0
\end{aligned}
\tag{29}
$$

$$
\begin{aligned}
&\quad end \\
&\quad end \\
&end \\
&T_{ATMR\ B\ Best\ Times} = T_{ABBT} = t_{ATMRBB\ TMR} + t_{ATMRBB\ TSR} \\
&T_{ATMR\ B\ Best\ Viable\ Times} = T_{ABBT}(1 : length(T_{ABBT}) - (3 - P_{loops}(n))) \\
&[T_{ATMR\ B\ Best\ Early}, b_3] = max(T_{ATMR\ B\ Best\ Viable\ Times}) \\
&[T_{ATMR\ B\ Best\ Late}, b_4] = min(T_{ATMR\ B\ Best\ Viable\ Times})
\end{aligned} \tag{30}
$$

It was predicted that AHR MIPS experiencing an AHR Type B Best - Late error would process a program faster than AHR Type B Best - Early. An AHR Type B Best - Late error causes AHR MIPS to transition from TMR to TSR later in the program, even after performing Type B error recovery, thus more of the program is performed in TMR mode. An AHR Type B Best - Early error has no effect on the original TMR to TSR transition point because the error is detected early enough in a program that effectively all that occurs is that Type B error recovery is performed and the program restarts execution. An AHR Type B Best - Early error produces a runtime that is slower than EF AHR because if the error does not cause the architecture to perform an additional program loop in TMR mode, then the Type B Best - Early runtime takes longer than EF AHR by the same amount of time it takes to perform Type B error recovery and re-process instructions up-to the point at which the error was detected. The predicted runtime speeds of the AHR MIPS Type B errors with respect to EF AHR were from fastest to slowest: AHR Type B Best - Late, EF AHR, and AHR Type B Best - Early.

If an AHR Type B Best - Early or Late error occurs and the program is short enough, AHR MIPS may not be able to transition from TMR to TSR mode. If

AHR MIPS does not transition to TSR mode due to a Type B Best - Early or Late error, then AHR MIPS processes the entire program in TMR mode using the TMR instructions. The corresponding Type B Best - Early & Late runtimes will be similar to each other and will be nearly equivalent to the runtime of TMR MIPS with a Type B Best error because each architecture would process the program using TMR instructions and perform the same error recovery. Depending on the structure of the program, the Type B Best - Late runtime may be faster than Type B Best - Early because the Early error is always detected at the first SW of the TMR program, but the Late error is detected at the SW that follows the creation of an SRP and contributes the least to the overall runtime. If the programs are too short to tolerate an AHR Type B Best error and AHR MIPS cannot transition to TSR mode, then the predicted runtime speeds of the AHR MIPS Type B Best errors with respect to EF AHR were from fastest to slowest: AHR Type B Best - Early or Late and then EF AHR.

AHR TMR Mode Type B Worst Errors: Both the AHR TMR mode Type B Worst - Early and Late scenarios require that the error is detected while writing the loop counter during creation of a TMR SRP. An AHR Type B Worst - Early error occurs while creating the first TMR SRP of the AHR program. An AHR Type B Worst - Late error occurs while creating the last TMR SRP of the AHR program. The AHR Type B Worst - Early error scenario causes the overall runtime of the AHR program to increase by the number of instructions processed up-to the creation of the first SRP, but does not affect the TMR to TSR transition point. The AHR Type B Worst - Late error scenario requires that the maximum amount of TMR instructions be re-processed and extends the TMR to TSR transition point. Figures 14 & 15 below depict what each AHR Type B Worst error variation looks like.

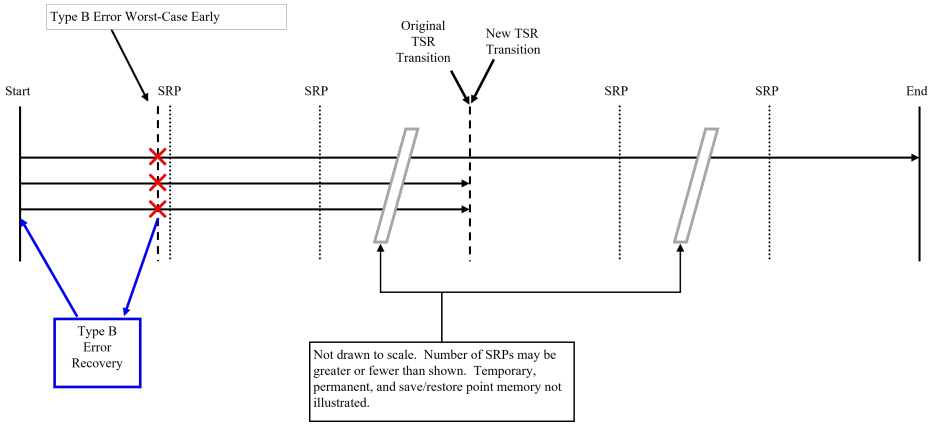

Figure 14. AHR MIPS TMR Type B Error Worst - Early [20]

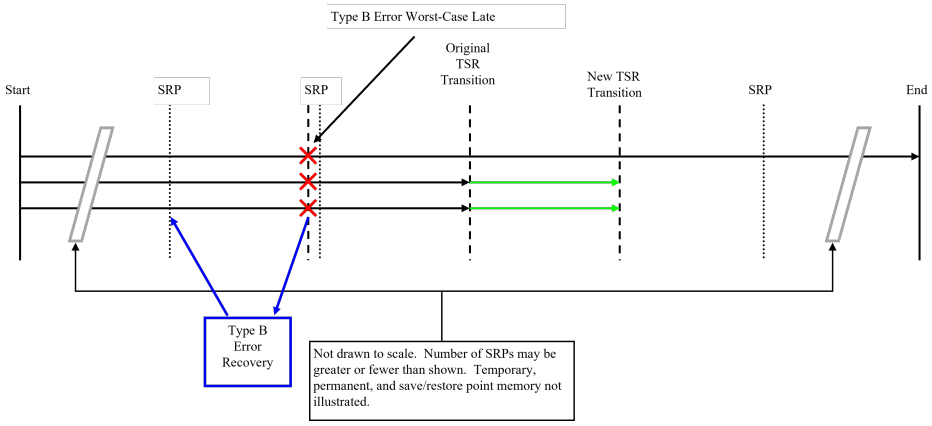

Figure 15. AHR MIPS TMR Type B Error Worst - Late [20]

The process used to calculate the runtime of any AHR MIPS program that experienced a TMR Type B Worst - Early error ($T_{ATMR\ B\ Worst\ Early}$) is given by Equations 31 and 32 [20], where $P_{loops\ TMR\ B\ Worst\ Early}$ is the new TMR to TSR transition point due to the detected error during creation of the first SRP, $n_{ASRP\ B\ Worst\ Early}$ is the number of TMR SRPs created before the transition, and SDT_{ATMR} is an array containing the save time differences between save indices that occur during the TMR portion of the AHR program. Equations 31 and 32 should be considered as one process, but were split due to formatting.

The values held in the SDT_{ATMR} array represent the calculated runtimes between consecutive TMR SRPs and are calculated using Equation 65 in Ch. 5 Section 5.4.1.3 of Hamilton's dissertation [20]. $P_{loops\ TMR\ B\ Worst\ Early}$ is calculated using Equation 66 in Ch. 5 Section 5.4.1.3 of Hamilton's dissertation [20]. The value of $n_{ASRP\ B\ Worst\ Early}$ is calculated using Equation 67 in Ch. 5 Section 5.4.1.3 of Hamilton's dissertation [20].

$T_{ATMR\ B\ Worst\ Early}$ is calculated by adding the time AHR MIPS spent in TMR mode and TSR mode due to a Type B Worst - Early error, represented by the variables $t_{ATMRBWE\ TMR}$ and $t_{ATMRBWE\ TSR}$ respectively. In Equation 31, $SDT_{ATMR}(2)$ is used because the Worst - Early error is detected while creating the first TMR SRP, so the time it takes to reach the first SRP, not including the default SRP created at program start, is the value held at $SDT_{ATMR}(2)$. To realize a Type B Worst - Early error in hardware, the error is always injected into the TMR registers that store the loop counter before creating the first SRP.

$$
\begin{aligned}
&t_{nom\ BWE} = t_{TMR\ init} + P_{loops\ TMR\ B\ Worst\ Early} \cdot T_{TMR\ loop} + \cdots \\
&T_{TMR\ SRP} \cdot n_{ASRP\ B\ Worst\ Early} \\
&t_{err\ BWE} = T_{TMR\ SRP\ Err} + T_{TMR\ recB} \\
&if\ P_{loops\ TMR\ B\ Worst\ Early} = 0 \\
&\quad t_{ATMRBWE\ TMR} = t_{nom\ BWE} + t_{err\ BWE} + SDT_{ATMR}(2) + t_{TMR \rightarrow TSR} \\
&\quad t_{ATMRBWE\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Worst\ Early}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + 2 \cdot T_{TSR\ SRP1} + T_{TSR\ BrI} - T_{TSR\ skip} \\
&elseif\ P_{loops\ TMR\ B\ Worst\ Early} = 1 \\
&\quad t_{ATMRBWE\ TMR} = t_{nom\ BWE} + t_{err\ BWE} + SDT_{ATMR}(2) + t_{TMR \rightarrow TSR} \\
&\quad t_{ATMRBWE\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Worst\ Early}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + T_{TSR\ SRP1} + T_{TSR\ BrI} - T_{TSR\ skip}
\end{aligned}
\tag{31}
$$

$$
\begin{aligned}
&elseif\ P_{loops\ TMR\ B\ Worst\ Early} = 2 \\
&\quad t_{ATMRBWE\ TMR} = t_{nom\ BWE} + t_{err\ BWE} + SDT_{ATMR}(2) + t_{TMR \rightarrow TSR} \\
&\quad t_{ATMRBWE\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Worst\ Early}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP1} + T_{TSR\ BrI} - \tfrac{2}{3} T_{TSR\ skip} \\
&else \\
&\quad t_{ATMRBWE\ TMR} = t_{TMR\ init} + n_{loops} \cdot T_{TMR\ loop} + \cdots \\
&\quad T_{TMR\ SRP} \cdot (n_{SRP} - 1) + t_{err\ BWE} + SDT_{ATMR}(2) + T_{TMR\ BrI} \\
&\quad t_{ATMRBWE\ TSR} = 0 \\
&end \\
&T_{ATMR\ B\ Worst\ Early} = t_{ATMRBWE\ TMR} + t_{ATMRBWE\ TSR}
\end{aligned}
\tag{32}
$$

The process used to calculate the runtime of any AHR MIPS program that experienced a TMR Type B Worst - Late error ($T_{ATMR\ B\ Worst\ Late}$) is given by Equations 33 and 34 [20], where $P_{loops\ TMR\ B\ Worst\ Late}$ is the new TMR to TSR transition point due to the detected error during creation of the last SRP and $n_{ASRP\ B\ Worst\ Late}$ is the number of TMR SRPs created before the transition. Equations 33 and 34 should be considered as one process, but were split due to formatting.

$P_{loops\ TMR\ B\ Worst\ Late}$ is calculated using Equation 70 in Ch. 5 Section 5.4.1.3 of Hamilton's dissertation [20]. The value of $n_{ASRP\ B\ Worst\ Early}$ is calculated using Equation 71 in Ch. 5 Section 5.4.1.3 of Hamilton's dissertation [20].

$T_{ATMR\ B\ Worst\ Late}$ is calculated by adding the time AHR MIPS spent in TMR mode and TSR mode due to a Type B Worst - Late error, represented by the variables $t_{ATMRBWL\ TMR}$ and $t_{ATMRBWL\ TSR}$ respectively. Throughout Equation 33, the SDT_{ATMR} array is indexed using $(length(SDT_{ATMR}) - (3 - P_{loops}(n)))$ to select the calculated time between creation of the last TMR SRP and the prior SRP, while also

accounting for the current n^{th} instruction set's original TMR to TSR transition point and the total number of loops performed. To realize a Type B Worst - Late error in hardware, the error is always injected into the TMR registers that store the loop counter before creating the last TMR SRP.

$$
\begin{aligned}
&t_{nom\ BWL} = t_{TMR\ init} + P_{loops\ TMR\ B\ Worst\ Late} \cdot T_{TMR\ loop} + \cdots \\
&T_{TMR\ SRP} \cdot n_{ASRP\ B\ Worst\ Late} \\
&t_{err\ BWL} = T_{TMR\ SRP\ Err} + T_{TMR\ recB} \\
&if\ P_{loops\ TMR\ B\ Worst\ Late} = 0 \\
&\quad t_{ATMRBWL\ TMR} = t_{nom\ BWL} + t_{err\ BWL} + t_{TMR \rightarrow TSR} + \cdots \\
&\quad SDT_{ATMR}(length(SDT_{ATMR}) - (3 - P_{loops}(n))) \\
&\quad t_{ATMRBWL\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Worst\ Late}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + 2 \cdot T_{TSR\ SRP1} + T_{TSR\ BrI} - T_{TSR\ skip} \\
&elseif\ P_{loops\ TMR\ B\ Worst\ Late} = 1 \\
&\quad t_{ATMRBWL\ TMR} = t_{nom\ BWL} + t_{err\ BWL} + t_{TMR \rightarrow TSR} + \cdots \\
&\quad SDT_{ATMR}(length(SDT_{ATMR}) - (3 - P_{loops}(n))) \\
&\quad t_{ATMRBWL\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Worst\ Late}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + T_{TSR\ SRP1} + T_{TSR\ BrI} - T_{TSR\ skip} \\
&elseif\ P_{loops\ TMR\ B\ Worst\ Late} = 2 \\
&\quad t_{ATMRBWL\ TMR} = t_{nom\ BWL} + t_{err\ BWL} + t_{TMR \rightarrow TSR} + \cdots \\
&\quad SDT_{ATMR}(length(SDT_{ATMR}) - (3 - P_{loops}(n))) \\
&\quad t_{ATMRBWL\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Worst\ Late}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP1} + T_{TSR\ BrI} - \tfrac{2}{3} T_{TSR\ skip}
\end{aligned}
\tag{33}
$$

$$
\begin{aligned}
&else \\
&\quad t_{ATMRBWL\ TMR} = t_{TMR\ init} + n_{loops} \cdot T_{TMR\ loop} + \cdots \\
&\quad T_{TMR\ SRP} \cdot (n_{SRP} - 1) + t_{err\ BWL} + \cdots \\
&\quad SDT_{ATMR}(length(SDT_{ATMR}) - (3 - P_{loops}(n))) + T_{TMR\ BrI} \\
&\quad t_{ATMRBWL\ TSR} = 0 \\
&end \\
&T_{ATMR\ B\ Worst\ Late} = t_{ATMRBWL\ TMR} + t_{ATMRBWL\ TSR}
\end{aligned}
\tag{34}
$$

It was predicted that AHR MIPS experiencing an AHR Type B Worst - Late error would process a program faster than AHR Type B Worst - Early. An AHR Type B Worst - Late error causes AHR MIPS to transition from TMR to TSR later in the program, even after performing Type B error recovery and re-processing instructions up-to creation of the SRP where the error was detected, thus more of the program is performed in TMR mode. An AHR Type B Worst - Early error has no effect on the original TMR to TSR transition point because the error is detected while creating the first SRP of the program and effectively all that occurs is that Type B error recovery is performed and the program restarts execution. An AHR Type B Worst - Early error produces a runtime that is slower than EF AHR because if the error does not cause the architecture to perform an additional program loop in TMR mode, then the Type B Worst - Early runtime takes longer than EF AHR by the same amount of time it takes to perform Type B error recovery and re-process instructions up-to creation of the SRP where the error was detected. The predicted runtime speeds of the AHR MIPS Type B errors with respect to EF AHR MIPS were from fastest to slowest: AHR Type B Worst - Late, EF AHR, AHR Type B Worst - Early.

The previous predictions were made for large programs based upon the results from the prior research [20]. For this research the programs only perform 3 loops. New predictions are necessary to capture the performance relationship between AHR

Type B Worst - Early & Late for the short programs used in this research. Specifically, if AHR MIPS can still transition from TMR to TSR mode after a Type B Worst - Early or Late error is injected, there may be only one TMR SRP created before transitioning and thus only one location where a Type B Worst error can be injected. In that case, the runtimes of both AHR Type B Worst - Early & Late are equivalent and will be slower than EF AHR MIPS.

For this research, if an AHR MIPS program is long enough to create two TMR SRPs then both an AHR Type B - Worst Early & Late error can be realized and will produce different results. If AHR MIPS does not transition to TSR mode due to a Type B Worst - Late error, then AHR MIPS processes the entire program in TMR mode using the TMR instructions. The corresponding Type B Worst - Late runtime will be nearly equivalent to the runtime of TMR MIPS with a Type B Worst error because both architectures would process the program using TMR instructions, perform the same Type B error recovery, and re-process instructions up-to creation of the SRP where the error was detected. In that case, where AHR MIPS cannot transition from TMR to TSR mode due to a Type B - Worst Late error, the predicted runtime speeds of the AHR MIPS Type B Worst errors with respect to EF AHR were from fastest to slowest: AHR Type B Worst - Late, EF AHR, and AHR Type B Worst - Early.

AHR TSR Mode Errors: After AHR MIPS transitions from TMR mode to TSR mode, only the TSR Best and TSR Worst error scenarios can occur. The AHR TSR Best & Worst error scenarios are realized in the same manner as was described for TSR MIPS. While in TSR mode errors are detected using a BNE instruction executed before a SW that writes the values in an EDDI-TSR register pair to memory.

An AHR TSR Best error occurs when an error is detected at the first BNE-SW pair of the TSR portion of the AHR instruction set or at the first BNE-SW pair after

a TSR SRP is created. For the short instruction sets used in this research, whether the error is detected at the first BNE-SW pair after TMR to TSR transition or after a TSR SRP has been made, the effect on the runtime is the same. An AHR TSR Worst error occurs when an error is detected either while writing the loop counter during TSR SRP creation or at the last BNE instruction of the program during the last program loop. The short instruction sets used in this research created a TSR SRP every loop, therefore there was never a case where detecting an error at the last BNE instruction of the program produced a runtime worse than detecting an error during SRP creation.

Both AHR TSR error scenarios increase the total runtime of an AHR program. Figures 16 & 17 below depict what each AHR TSR error variation looks like as applied to a generic AHR program.

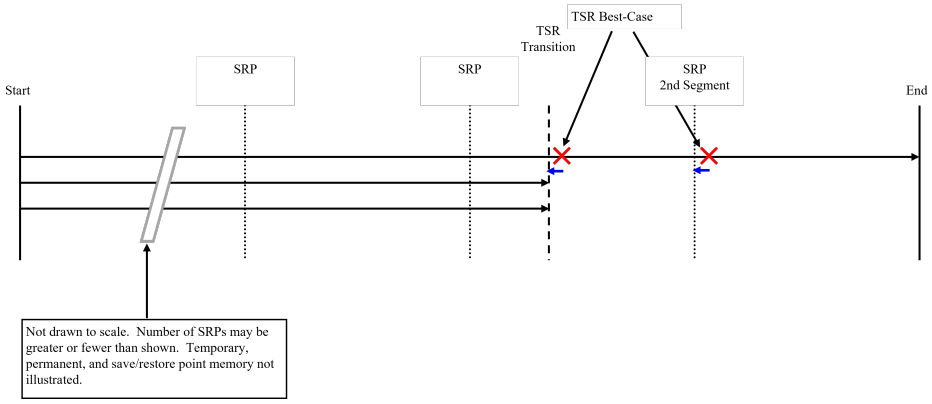

Figure 16. AHR MIPS TSR Error - Best [20]

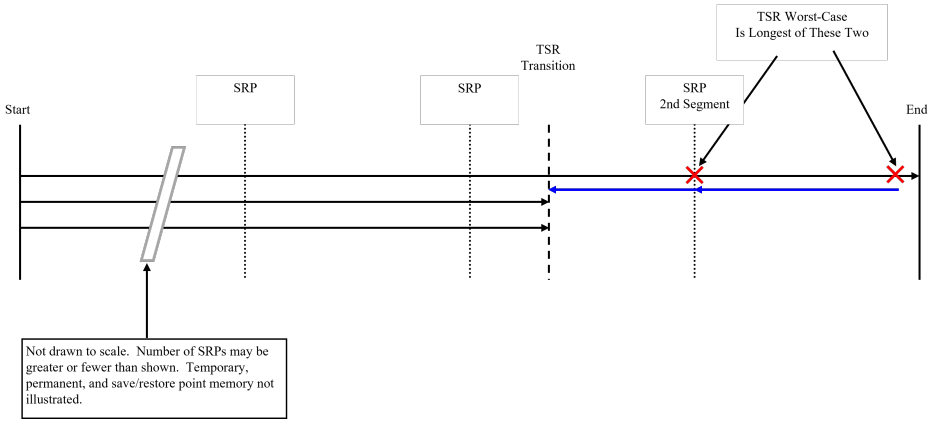

Figure 17. AHR MIPS TSR Error - Worst [20]

The runtime of any AHR MIPS program that experienced a TSR Best-case error ($T_{ATSR\ Best}$) is given by Equation 35 [20] where $T_{EF\ AHR\ MIPS}$ is calculated using Equation 21, $t_{AHR\ TMR}$ & $t_{AHR\ TSR}$ are calculated using Equation 21, $T_{TSR\ Rec}$ is the time it took to perform TSR error recovery, and $T_{TSR\ ret}$ is calculated using Equation 19. To realize a TSR Best error in hardware, the error is always injected into one of the source registers evaluated by the first BNE-SW pair of the TSR portion of the AHR program.

$$\begin{aligned} T_{ATSR\ Best} &= T_{EF\ AHR\ MIPS} + T_{TSR\ Rec} + T_{TSR\ ret} \\ T_{ATSR\ Best} &= t_{AHR\ TMR} + t_{AHR\ TSR} + T_{TSR\ Rec} + T_{TSR\ ret} \\ t_{ATSRB\ TMR} &= t_{AHR\ TMR} \\ t_{ATSRB\ TSR} &= t_{AHR\ TSR} + T_{TSR\ Rec} + T_{TSR\ ret} \\ T_{ATSR\ Best} &= t_{ATSRB\ TMR} + t_{ATSRB\ TSR} \end{aligned} \tag{35}$$

The process used to calculate the runtime of any AHR MIPS program that experienced a TSR Worst-case error is given by Equation 36 [20].

The total AHR TSR Worst error runtime ($T_{ATSR\ Worst}$) is calculated by adding the amount of time AHR MIPS operated in TMR mode and TSR mode, represented by $t_{ATSRW\ TMR}$ and $t_{ATSRW\ TSR}$ respectively. The amount of time spent in either

mode is dependent on the TMR to TSR transition point P_{loops}. Equation 36 uses an if statement dependent upon the value of P_{loops} to calculate the AHR TSR Worst error runtime. The value for $t_{ATSRW\ TMR}$ was calculated as part of Equation 21, where $t_{ATSRW\ TMR} = t_{AHR\ TMR}$. The value for $t_{AHR\ TSR}$ was calculated as part of Equation 21. A TSR Worst-case error is always realized in hardware by detecting an error while writing the last register containing the loop counter during the SRP creation process, but the loop count at which the error should be injected depends on P_{loops}.

When $P_{loops} = 1$ the program completes two TSR loops and creates two TSR SRPs. The Worst-case error can either occur while writing TSR SRP1 or TSR SRP0. The extra runtime caused by an error while writing SRP1 is calculated and represented by the variable $atsrw_1$. The extra runtime caused by an error while writing SRP0 is calculated and represented by the variable $atsrw_2$. Once those values are calculated an if statement is used to determine which value is larger, or which value represents the overall TSR Worst-case error. If $atsrw_1$ was larger, then the error should be injected/detected when the program loop count is two. If $atsrw_2$ was larger, then the error should be injected/detected when the program loop count is one.

When $P_{loops} = 2$ the program completes only one TSR loop. In this case, the TSR Worst-case error is always detected during creation of the only TSR SRP that is made. The extra runtime caused by an error while writing SRP1 is calculated and represented by the variable $atsrw_3$. If $P_{loops} = 2$ then only one TSR SRP is created, and the error should be injected/detected when the program loop count is one.

When $P_{loops} > 2$ the generated program was too short and never transitioned into TSR mode, therefore no TSR Worst error can be injected and the overall runtime is just the error free TMR runtime ($t_{AHR\ TMR}$).

$$
\begin{aligned}
&if\ P_{loops} = 0 \\
&\quad t_{ATSRW\ TMR} = t_{AHR\ TMR} \\
&\quad t_{ATSRW\ TSR} = t_{AHR\ TSR} + T_{TSR\ Rec} + \cdots \\
&\quad T_{TSR\ loop} + T_{TSR\ SRP1\ Err} \\
&elseif\ P_{loops} = 1 \\
&\quad atsrw_1 = T_{TSR\ Rec} + T_{TSR\ loop} - \cdots \\
&\quad \tfrac{1}{3}T_{TSR\ skip} + T_{TSR\ SRP1\ Err} \\
&\quad atsrw_2 = T_{TSR\ Rec} + T_{TSR\ loop} - \cdots \\
&\quad \tfrac{2}{3}T_{TSR\ skip} + T_{TSR\ SRP0\ Err} \\
&\quad t_{ATSRW\ TMR} = t_{AHR\ TMR} \\
&\quad if\ atsrw_1 > atsrw_2 \\
&\qquad t_{ATSRW\ TSR} = t_{AHR\ TSR} + atsrw_1 \\
&\quad else \\
&\qquad t_{ATSRW\ TSR} = t_{AHR\ TSR} + atsrw_2 \\
&\quad end \\
&elseif\ P_{loops} = 2 \\
&\quad atsrw_3 = T_{TSR\ Rec} + (3 - P_{loops}) \cdot T_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP1\ Err} - \tfrac{2}{3}T_{skip} \\
&\quad t_{ATSRW\ TMR} = t_{AHR\ TMR} \\
&\quad t_{ATSRW\ TSR} = t_{AHR\ TSR} + atsrw_3 \\
&else \\
&\quad t_{ATSRW\ TMR} = t_{AHR\ TMR} \\
&\quad t_{ATSRW\ TSR} = 0 \\
&end \\
&T_{ATSR\ Worst} = t_{ATSRW\ TMR} + t_{ATSRW\ TSR}
\end{aligned}
\tag{36}
$$

It was predicted that AHR MIPS experiencing an AHR TSR Best error would process a program faster than an AHR TSR Worst error. An AHR TSR Best error is detected at the first BNE-SW pair of the TSR portion of the AHR program, and only increases the runtime by the time it takes to perform TSR error recovery and re-process instructions up-to the point at which the error was detected. An AHR TSR Worst error is detected while writing a SRP and requires that an additional TSR loop must be performed. Since both error recovery and an additional loop must be performed it was predicted that an AHR TSR Worst error produces program runtimes slower than EF TSR MIPS. Both AHR TSR errors perform slower than EF AHR because additional TSR instructions must be processed. The predicted runtime speeds of the AHR MIPS TSR errors with respect to EF AHR and EF TSR were from fastest to slowest: EF AHR, AHR TSR Best error, EF TSR, and AHR TSR Worst error.

3.5.2.4 Predicted Order of Runtime Performance

The list below rank orders the predicted/expected performance of each MIPS architecture processing the short programs generated for this research for all operating modes discussed in Section 3.5.2. The list ranks performance in terms of program runtime from fastest to slowest. The ranking assumes that AHR MIPS does not transition to TSR mode after experiencing a Type A Early, Type A Late, Type B Best - Late, and Type B Worst error. These predictions were discussed throughout Section 3.5.2 and potential differences to the rank order listed below were discussed as well as the error scenarios that could produce runtimes nearly equivalent to others.

1. EF TMR
2. TMR Type A

3. AHR Type A Early
4. AHR Type A Late
5. TMR Type B Best
6. AHR Type B Best - Late
7. TMR Type B Worst
8. EF AHR
9. AHR Type B Best - Early
10. AHR Type B Worst - Early
11. AHR Type B Worst - Late
12. AHR TSR Best Error
13. EF TSR
14. AHR TSR Worst Error
15. TSR Best Error
16. TSR Worst Error

3.5.2.5 Summary of MIPS Archs. & Operational Modes Measured

Timing equations were developed to calculate the runtime of a program processed by the TMR, TSR, and AHR MIPS architectures. Each architecture responds to simulated radiation-induced errors in different ways. TMR MIPS can operate in 4 modes: error free, Type A error, Type B Best error, and Type B Worst error. The runtime of a program processed by TMR MIPS in each operational mode was

calculated using the equations developed in Section 3.5.2.1. TSR MIPS can operate in 3 modes: error free, Best-case error, and Worst-case error. The runtime of a program processed by TSR MIPS in each operational mode was calculated using equations developed in Section 3.5.2.2. AHR MIPS processes programs in both TMR and TSR mode. Therefore, AHR MIPS can not only experience all the errors that each constituent architecture can, but also has new error scenarios classified as Early or Late. AHR MIPS can operate in 9 modes: error free, Type A Early, Type A Late, Type B Best - Early, Type B Best - Late, Type B Worst - Early, Type B Worst - Late, TSR Early/Best error, and TSR Late/Worst error. The runtime of a program processed by AHR MIPS in each operational mode was calculated using equations developed in Section 3.5.2.3. Every operational mode for each architecture was measured in hardware and compared against the results produced by the timing equations.

3.5.3 Measurement Methodology

The AHR MIPS hardware design used two SP701 Spartan 7 development boards. The first board (Board #1) contained the AHR MIPS processor, MIBO, MIBI, and LED oscillator components. The second board (Board #2) contained the Memory, MEBI, and MEBO components. A Keysight DSOSO54A Digital Storage Oscilloscope [51] was connected to Board #1 to measure program runtimes. The oscilloscope was connected to a ground pin on Board #1 and specifically measured the output of the LED oscillator that was routed to an on-board PMOD pin. A diagram of the experimental setup is shown in Figure 18 below.

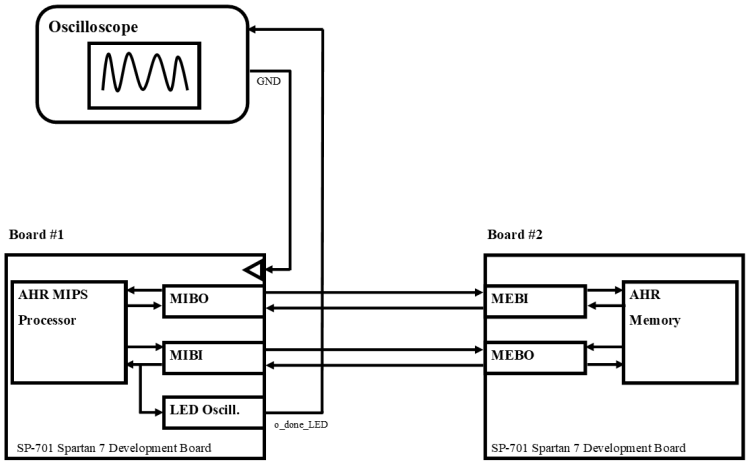

Figure 18. Experimental Setup of AHR MIPS in Hardware

An image of the experimental setup connected to the oscilloscope is shown in Figure 19.

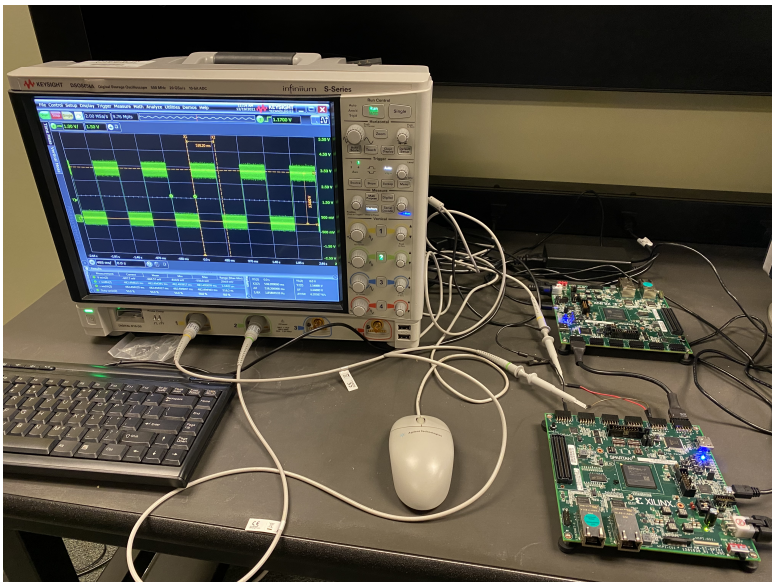

Figure 19. AHR MIPS Hardware Test Setup

An image providing a close-up view of Boards #1 & #2, the connection between them, and the measurement probe connections to Board #1 is shown in Figure 20.

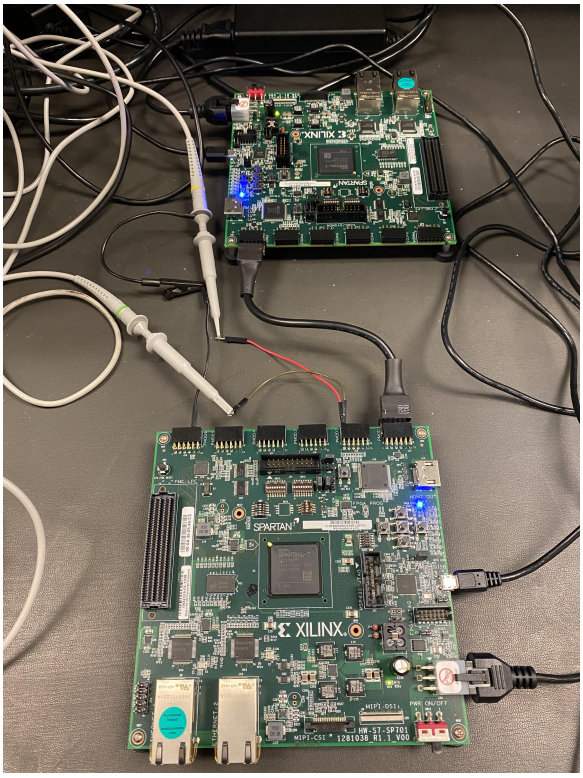

Figure 20. AHR MIPS Boards #1 & #2

Table 6 describes how the PMOD pins on each SP701 Spartan 7 development board were configured to enable inter-board communications.

Table 6. SP-701 PMOD Pin Assignments for Inter-Board Comms. & Measurements

Board #	**PMOD Pin**	**Signal Name**
Board #1	D14	i_MIBI_RX
	C13	i_MIBO_RX
	C14	o_MIBI_TX
	A13	o_MIBO_TX
	D16	o_DONE_M
Board #2	C13	i_MEBI_RX
	D14	i_MEBO_RX
	A13	o_MEBI_TX
	C14	o_MEBO_TX

The oscilloscope user manual did not provide any steps for calibration, so all measurements were made assuming that the oscilloscope was functioning within specifications [51]. Every MIPS architecture and program implemented on the FPGAs were allowed to operate for one minute before collecting hardware runtime data to allow the temperature of each device to settle after programming. The hardware runtime of each program was measured 10 consecutive times and used to calculate an average hardware runtime. The calculated runtime of each program was used to set the oscilloscope timescale to be large enough to capture 10 consecutive hardware program runs. The oscilloscope was also configured to provide a real-time runtime measurement between the rise and fall of one DONE signal and output that value to the screen. This value was used during each runtime data measurement to quickly determine whether the architecture & program combination behaved as predicted by the runtime calculation.

Any program runtime measured in hardware was expected to be longer than the calculated runtime due to signal propagation delays. For the longest program processed by the slowest architecture & operating mode (TSR MIPS Worst Error) the difference between the measured runtime and calculated runtime was roughly $50\mu s$. For the shortest program processed by the fastest architecture & operating mode (EF TMR MIPS) the difference between the measured runtime and calculated runtime was roughly $15\mu s$. These observed differences provided the upper and lower limits for the total signal propagation delay that should accumulate during execution of one of the programs used in this research. Having established the upper and lower limits for total signal propagation delay, if the difference between the measured program runtime and the calculated runtime was less than $50\mu s$, then that measurement was considered valid, and the data was collected. If the difference between the measured runtime and calculated runtime was greater than $50\mu s$ or less than the calculated value, then the measurement was considered invalid. An invalid measurement indicated that the difference between the measured runtime and calculated runtime could not be solely attributed to accumulated signal propagation delays and would require further investigation to determine the cause of the discrepancy. Invalid runtime measurement data was not collected. Every valid program runtime measurement was saved to a unique file, and the average hardware runtime was calculated as part of post-processing using that file.

3.5.4 Summary of Hardware Testing and Data Collection

The AHR MIPS architecture was modified as described in Section 3.5.1 to produce PROGRAM DONE signals that could be measured by an oscilloscope to determine the runtime of programs in hardware.

20 out of 100 randomly generated instruction sets were tested in hardware for each

architecture and for each operational mode described in Section 3.5.2. To test each instruction set a Board #1 and Board #2 hardware project were generated using the Xilinx Vivado software.

Board #2 always contained the memory, MEBI, and MEBO components. Errors were never injected into Board #2 to realize the various error scenarios for each MIPS architecture. Since Board #2 was never modified, regardless of the error scenario tested, only 20 Board #2 hardware projects were generated for each MIPS architecture tested.

Board #1 contained the specific MIPS architecture, MIBO, and MIBI, but since all the various operational modes are realized by injecting errors into the MIPS processor, this required that a unique hardware project be generated for every operational mode. This research tested the following operational modes: three error free (one for each MIPS processor), three TMR MIPS error scenarios, two TSR MIPS error scenarios, and eight AHR MIPS error scenarios. Thus, to test all 20 instructions sets, a total of 60 Board #2 and 320 Board #1 hardware projects were generated and measured in hardware.

Each program was repeatedly processed in hardware, and the oscilloscope was used to capture 10 consecutive program runs which were measured using the output DONE signal from Board #1. The program run captures were then post-processed to calculate a final average runtime for each program. These final averages were compared against what was calculated using the equations developed in Section 3.5.2 to not only verify that the hardware implementation of AHR MIPS is correct and predictable, but also that AHR MIPS exhibits the same operational behaviors seen in the results of the previous research [20].

3.6 Ch. III Summary

This chapter discussed the rationale behind using two Xilinx Spartan 7 FPGAs as the implementation targets for the new AHR MIPS architecture. This chapter described how AHR MIPS was modified to be successfully implemented in hardware, which focused on developing the communications path necessary to support inter-board communication. This chapter defined the parameters of the instruction sets generated and used to test the operation of TMR, TSR, and AHR MIPS in hardware. The process used to collect the timing information of instructions and non-instruction processes necessary to predict/calculate program runtimes was also described. Finally, this chapter defined the operational modes tested on each MIPS architecture, described how the runtime of programs under error free and various error scenarios were calculated, and discussed the methodology employed to collect hardware program runtime measurements. The next chapter will discuss the results of hardware testing and compare the collected program runtime results against those that were calculated.

IV. Results

4.1 Introduction

This chapter discusses the runtime performance results of the error free and error prone TMR, TSR, and AHR MIPS architectures. These results compare the runtime of programs on each architecture under various operating conditions against the measured program runtimes on each architecture implemented in hardware. The equations used to calculate the runtime of a given program operating on an architecture under various operating conditions were discussed in Ch. III Section 3.5.2. The methodology used to measure the runtime of a program in hardware was described in Ch. III Section 3.5.3. This chapter provides analyses that demonstrate AHR MIPS can successfully be implemented on a COTS FPGA, and that the runtime performance of AHR MIPS under various operating conditions in hardware matches what was predicted in not only this research but in the previous AHR research as well [20].

Section 4.2 discusses the runtime results that were calculated and measured in hardware for every operating mode of TMR MIPS. Section 4.3 discusses the runtime results that were calculated and measured in hardware for every operating mode of TSR MIPS. Section 4.4 discusses the runtime results that were calculated and measured in hardware for every operating mode of AHR MIPS. Section 4.5 describes the challenges that were encountered during hardware measurements and how those challenges affected the collected results. Section 4.6 compares the performance of AHR MIPS against TMR MIPS and TSR MIPS, and analyzes each AHR MIPS error scenario to determine whether the performance matched predictions and expectations.

4.2 TMR MIPS Results

TMR MIPS was simulated and tested under four operating modes: Error Free, Type A Error, Type B - Best, and Type B - Worst. The TMR MIPS versions of the 20 randomly generated instruction sets were processed by the TMR MIPS architecture under each operating mode. The results of TMR MIPS operating without error is presented in Section 4.2.1. The results of TMR MIPS operating while a Type A, Type B - Best, or Type B - Worst error were injected is presented in Section 4.2.2.

4.2.1 Error Free TMR MIPS Results

The program runtime results that were calculated and measured for each of the 20 TMR MIPS instruction sets processed by an Error Free (EF) TMR MIPS architecture are shown in Figure 21. Plots like the one presented in Figure 21 are used throughout this chapter. The X-Axis represents the "Instruction Set #" and merely indicates which 20 instruction sets out of the 100 randomly generated were measured in hardware. The "Instruction Set #" itself is arbitrary and has no effect on performance. The Y-Axis presents the measured runtime of an instruction set/program in milli-seconds (ms). A calculated value always exists for every instruction set and the calculated runtime is indicated using a red "x". The measured runtime of an instruction set is indicated using a green dot ".". The measured values are connected via lines to better distinguish the values measured from the values calculated. The differences between the error free program runtimes that were calculated and those measured cannot be seen in this figure.

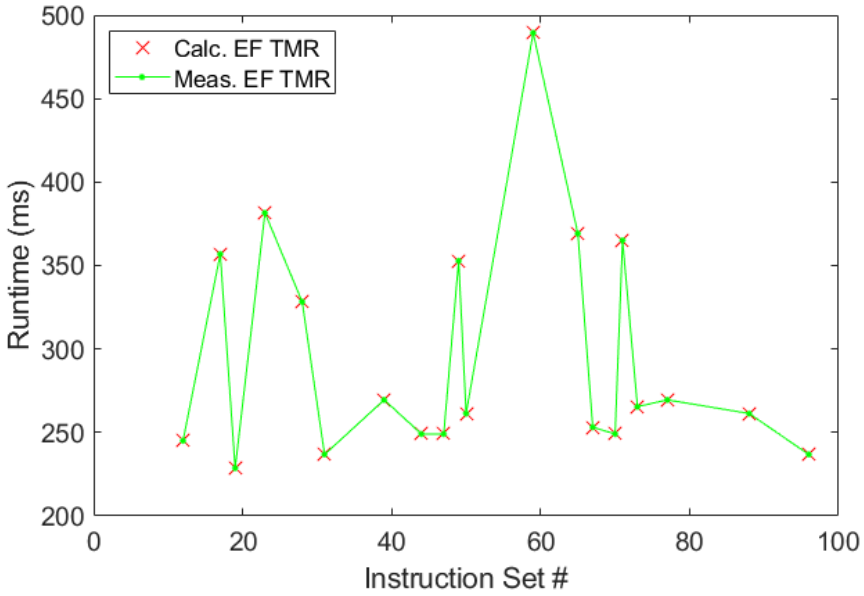

Figure 21. Calculated & Measured Runtime Performance of Error-Free TMR MIPS

The average percent difference between the error free program runtimes that were calculated and those measured was found using Equation 37 [20].

$$PD_{Time\ TMR\ EF} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{TMR\ EF\ Meas} - T_{TMR\ EF\ Calc}}{T_{TMR\ EF\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{37}$$

The average percent difference is:

$$PD_{Time\ TMR\ EF} = 1.301 \cdot 10^{-3}\% \tag{38}$$

The average percent difference indicates that the equations used to calculate the runtime performance of EF TMR MIPS closely predict the runtime that was measured in hardware. The average percent difference is non-zero because the new TMR MIPS architecture is distributed across two Spartan 7 FPGA boards that are connected using wires, and thus the runtime measurements made in hardware are expected to

be higher due to signal propagation delays and clock mismatches between the two boards.

4.2.2 Error Prone TMR MIPS Results

4.2.2.1 TMR MIPS Type A Error

The program runtime results that were calculated and measured for each of the 20 TMR MIPS instruction sets processed by a TMR MIPS architecture experiencing a Type A error are shown in Figure 22. The differences between the Type A program runtimes that were calculated and those measured cannot be seen in this figure.

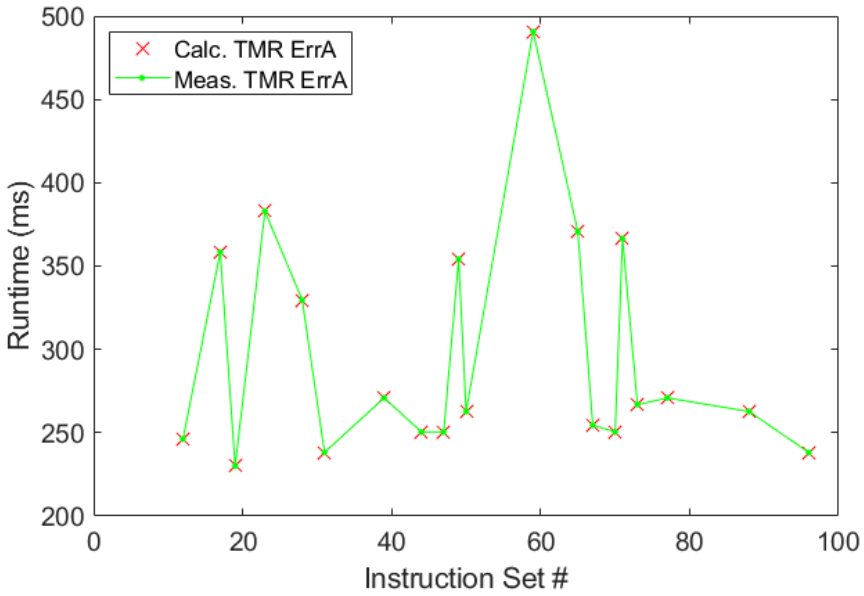

Figure 22. Calculated & Measured TMR ErrA Runtimes

The average percent difference between the Type A program runtimes that were calculated and those measured was found using Equation 39.

$$PD_{Time\ TMR\ ErrA} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{TMR\ ErrA\ Meas} - T_{TMR\ ErrA\ Calc}}{T_{TMR\ ErrA\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{39}$$

The average percent difference is:

$$PD_{Time\ TMR\ ErrA} = 1.301 \cdot 10^{-3}\% \tag{40}$$

The average percent difference indicates that the equations used to calculate the runtime performance of TMR MIPS with a Type A Error closely predict the runtime that was measured in hardware. The average percent difference is non-zero due to signal propagation delays.

4.2.2.2 TMR MIPS Type B Errors

The runtime results for the 20 instruction sets processed by a TMR MIPS architecture experiencing a Type B - Best or Type B - Worst error are shown on the same plot in Figure 23. Calculated results are shown for each of the 20 instruction sets for both the Type B - Best & Worst scenarios, but only 19 hardware measurements for each scenario were within $50\mu s$ of the calculated result and were considered valid. The invalid measurements were not collected because the measured runtimes were either faster than the calculated value or had a difference greater than $50\mu s$. These measurement-to-calculation runtime discrepancies were due to implementation issues that will be discussed in Section 4.5. For the remainder of Ch. IV, if a hardware result was considered invalid and was not collected, the line connecting measured points in figures like Figure 23 will be broken to indicate the absence of a measured runtime. The differences between the Type B - Best or Worst program runtimes that were calculated and those measured cannot be seen in this figure.

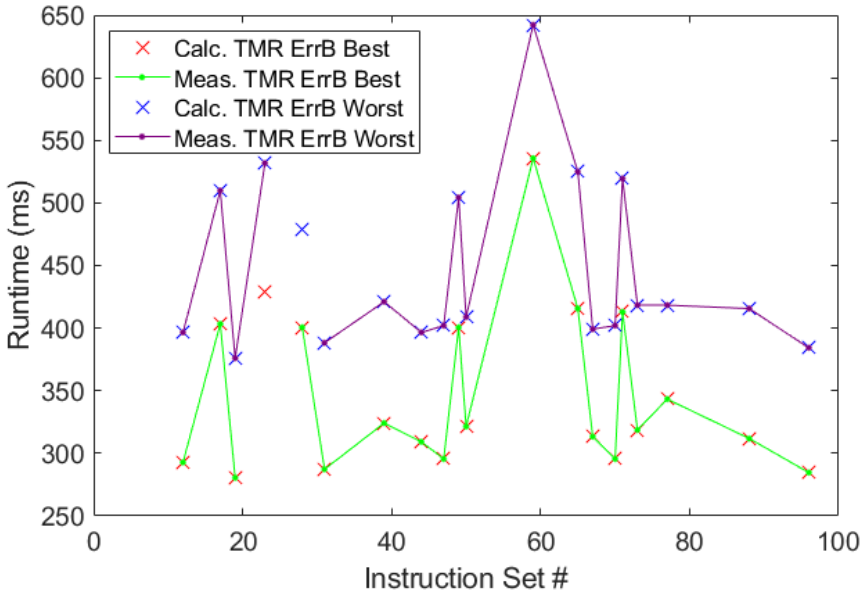

Figure 23. Calculated & Measured TMR ErrB Best and Worst Runtimes

The average percent differences between the Type B - Best or Worst program runtimes that were calculated and those measured were found using Equations 41 & 42.

$$PD_{Time\ TMR\ ErrBB} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{TMR\ ErrBB\ Meas} - T_{TMR\ ErrBB\ Calc}}{T_{TMR\ ErrBB\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{41}$$

$$PD_{Time\ TMR\ ErrBW} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{TMR\ ErrBW\ Meas} - T_{TMR\ ErrBW\ Calc}}{T_{TMR\ ErrBW\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{42}$$

The average percent difference for TMR Type B - Best is:

$$PD_{Time\ TMR\ ErrBB} = 1.246 \cdot 10^{-3}\% \tag{43}$$

The average percent difference for TMR Type B - Worst is:

$$PD_{Time\ TMR\ ErrBW} = 1.265 \cdot 10^{-3}\% \tag{44}$$

The average percent differences indicate that the equations used to calculate the runtime performance of TMR MIPS with a Type B - Best or Worst Error closely predict the runtime that was measured in hardware. The average percent difference is non-zero due to signal propagation delays.

4.3 TSR MIPS Results

TSR MIPS was simulated and tested under three operating modes: Error Free, Best Single Error, and Worst Single Error. The TSR MIPS versions of the 20 randomly generated instruction sets were processed by the TSR MIPS architecture under each operating mode. The results of TSR MIPS operating without error is presented in Section 4.3.1. The results of TSR MIPS operating while a Best case error or Worst case error were injected is presented in Section 4.3.2.

4.3.1 Error Free TSR MIPS Results

The program runtime results that were calculated and measured for each of the 20 TSR MIPS instruction sets processed by an EF TSR MIPS architecture are shown in Figure 24. The differences between the EF program runtimes that were calculated and those measured cannot be seen in this figure.

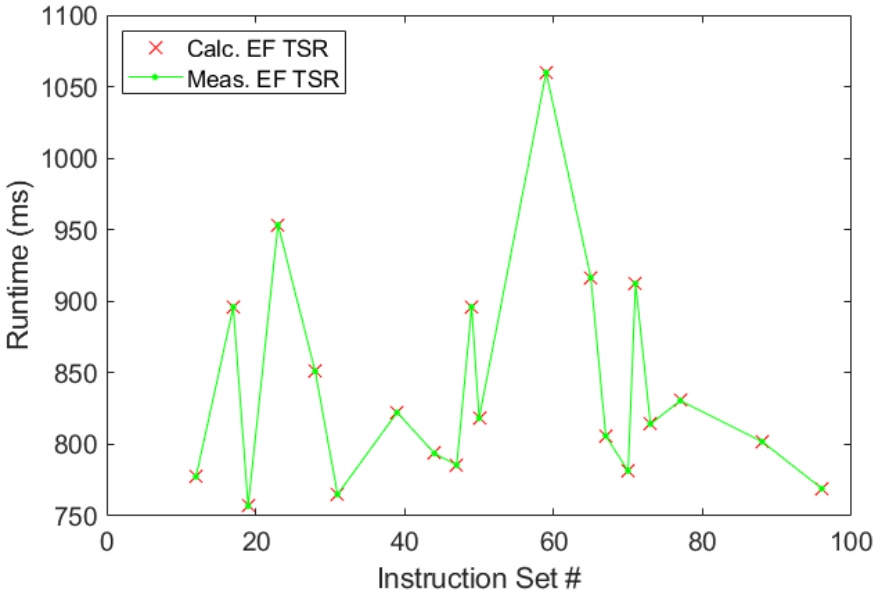

Figure 24. Calculated & Measured Runtime Performance of Error-Free TSR MIPS

The average percent difference between the EF program runtimes that were calculated and those measured was found using Equation 45 [20].

$$PD_{Time\ TSR\ EF} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{TSR\ EF\ Meas} - T_{TSR\ EF\ Calc}}{T_{TSR\ EF\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{45}$$

The average percent difference is:

$$PD_{Time\ TSR\ EF} = 1.296 \cdot 10^{-3}\% \tag{46}$$

The average percent difference indicates that the equations used to calculate the runtime performance of EF TSR MIPS closely predict the runtime that was measured in hardware. The average percent difference is non-zero due to signal propagation delays.

4.3.2 Error Prone TSR MIPS Results

The runtime results for the 20 instruction sets processed by a TSR MIPS architecture experiencing a Best case error or Worst case error are shown on the same plot in Figure 25. The differences between the Best or Worst program runtimes that were calculated and those measured cannot be seen in this figure.

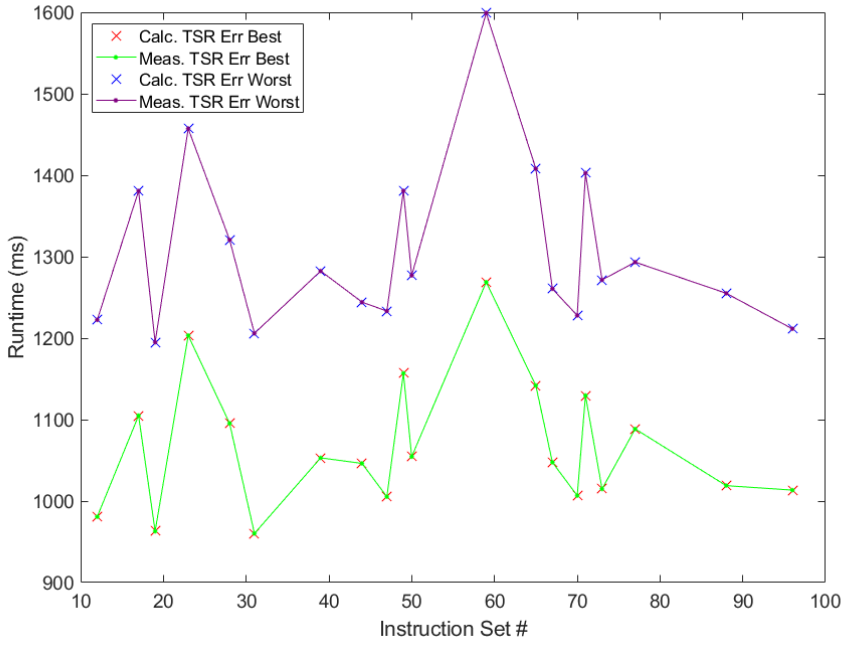

Figure 25. Calculated & Measured TSR Err Best and Worst Runtimes

The average percent differences between the Best Error or Worst Error program runtimes that were calculated and those measured were found using Equations 47 & 48.

$$PD_{Time\ TSR\ ErrB} = \cdots \frac{\sum_{n=1}^{Nprograms}\left[\frac{T_{TSR\ ErrB\ Meas}-T_{TSR\ ErrB\ Calc}}{T_{TSR\ ErrB\ Calc}}\cdot 100\%\right]}{N_{programs}} \tag{47}$$

$$PD_{Time\ TSR\ ErrW} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{TSR\ ErrW\ Meas} - T_{TSR\ ErrW\ Calc}}{T_{TSR\ ErrW\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{48}$$

The average percent difference for TSR Best Error is:

$$PD_{Time\ TSR\ ErrB} = 1.365 \cdot 10^{-3}\% \tag{49}$$

The average percent difference for TSR Worst Error is:

$$PD_{Time\ TSR\ ErrW} = 1.371 \cdot 10^{-3}\% \tag{50}$$

The average percent differences indicate that the equations used to calculate the runtime performance of TSR MIPS with a Best- or Worst-case error closely predict the runtime that was measured in hardware. The average percent difference is non-zero due to signal propagation delays.

4.4 AHR MIPS Results

AHR MIPS was simulated and tested under eight operating modes: EF, TMR Type A - Early, TMR Type A - Late, TMR Type B - Best Early, TMR Type B - Best Late, TMR Type B - Worst Early, TMR Type B - Worst Late, TSR Error Early/Best, and TSR Error Late/Worst. The AHR MIPS versions of the 20 randomly generated instruction sets were processed by the AHR MIPS architecture under each operating mode. The results of AHR MIPS operating without error is presented in Section 4.4.1. The results of AHR MIPS operating while the previously described errors were injected is presented in Section 4.4.2.

4.4.1 Error Free AHR MIPS Results

The program runtime results that were calculated and measured for each of the 20 AHR MIPS instruction sets processed by an EF AHR MIPS architecture are shown in Figure 26. The differences between the EF program runtimes that were calculated and those measured cannot be seen in this figure.

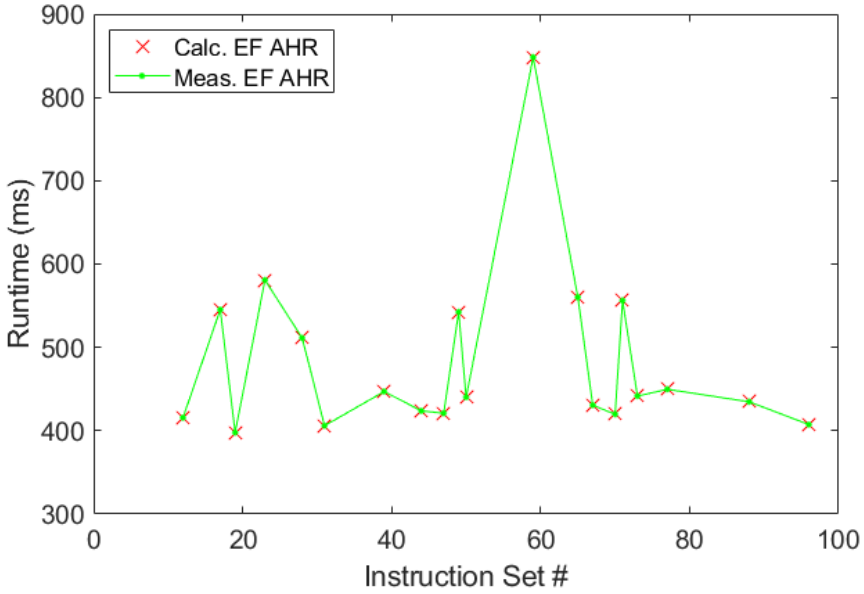

Figure 26. Calculated & Measured Runtime Performance of Error-Free AHR MIPS

The average percent difference between the EF program runtimes that were calculated and those measured were found using Equation 51 [20].

$$PD_{Time\ AHR\ EF} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{AHR\ EF\ Meas} - T_{AHR\ EF\ Calc}}{T_{AHR\ EF\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{51}$$

The average percent difference is:

$$PD_{Time\ AHR\ EF} = 1.264 \cdot 10^{-3}\% \tag{52}$$

The average percent difference indicates that the equations used to calculate the runtime performance of EF AHR MIPS closely predict the runtime that was measured in hardware. The average percent difference is non-zero due to signal propagation delays.

4.4.2 Error Prone AHR MIPS Results

4.4.2.1 AHR TMR MIPS Type A Errors

The runtime results for the 20 instruction sets processed by an AHR MIPS architecture experiencing a TMR Type A Early or TMR Type A Late error are shown on the same plot in Figure 27. Calculated results are shown for each of the 20 instruction sets for both the Type A Early & Late scenarios, but only 18 measurements were within $50\mu s$ of the calculated result and were considered valid for the Type A Early scenario. The invalid measurements were not collected because the measured runtimes were either faster than the calculated value or had a difference greater than $50\mu s$. These measurement-to-calculation runtime discrepancies were due to implementation issues that will be discussed in Section 4.5. Figure 27 presents calculated and measured data points that overlap for both error scenarios. To help distinguish between overlapping results, the calculated AHR Type A Early error runtimes are marked using a red circle "o". The differences between the Best or Worst program runtimes that were calculated and those measured cannot be seen in this figure.

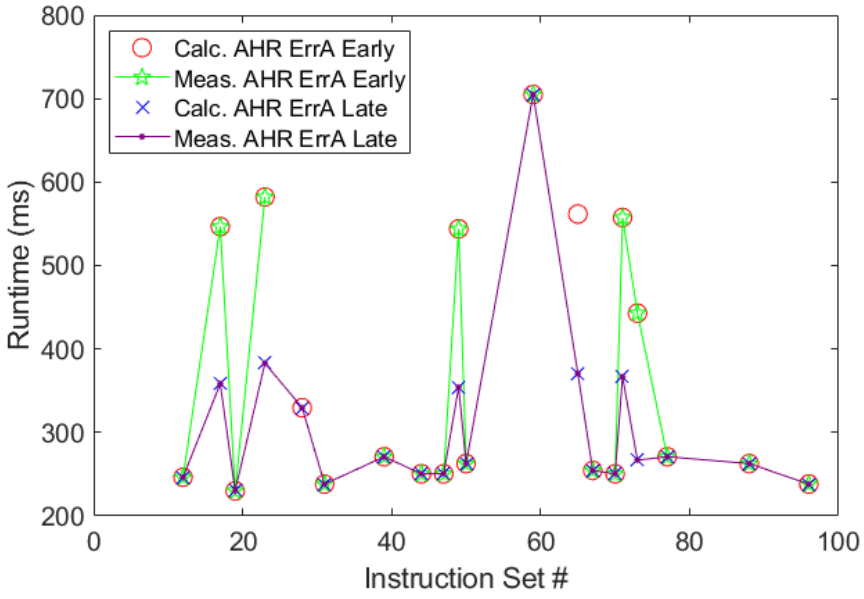

Figure 27. Calculated & Measured AHR TMR ErrA Early & Late Runtimes

The average percent differences between the AHR TMR Type A Early or AHR TMR Type A Late error program runtimes that were calculated and those measured were found using Equations 53 & 54.

$$PD_{Time\ AHR\ ErrAE} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{AHR\ ErrAE\ Meas} - T_{AHR\ ErrAE\ Calc}}{T_{AHR\ ErrAE\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{53}$$

$$PD_{Time\ AHR\ ErrAL} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{AHR\ ErrAL\ Meas} - T_{AHR\ ErrAL\ Calc}}{T_{AHR\ ErrAL\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{54}$$

The average percent difference for AHR Type A Early is:

$$PD_{Time\ AHR\ ErrAE} = 1.316 \cdot 10^{-3}\% \tag{55}$$

The average percent difference for AHR Type A Late is:

$$PD_{Time\ AHR\ ErrAL} = 1.312 \cdot 10^{-3}\% \tag{56}$$

The average percent differences indicate that the equations used to calculate the runtime performance of AHR MIPS with a TMR Type A Early or TMR Type A Late error closely predict the runtime that was measured in hardware. The average percent difference is non-zero due to signal propagation delays.

4.4.2.2 AHR TMR MIPS Type B Best Errors

The runtime results for the 20 instruction sets processed by an AHR MIPS architecture experiencing a TMR Type B Best - Early or TMR Type B Best - Late error are shown on the same plot in Figure 28. Calculated results are shown for each of the 20 instruction sets for both the Type B Best - Early & Late scenarios, but only 13 sets and 19 sets, respectively, had measured runtimes within $50\mu s$ of the calculated result and were considered valid. The invalid measurements were not collected because the measured runtimes were either faster than the calculated value or had a difference greater than $50\mu s$. These measurement-to-calculation runtime discrepancies were due to implementation issues that will be discussed in Section 4.5. The differences between the Best or Worst program runtimes that were calculated and those measured cannot be seen in this figure.

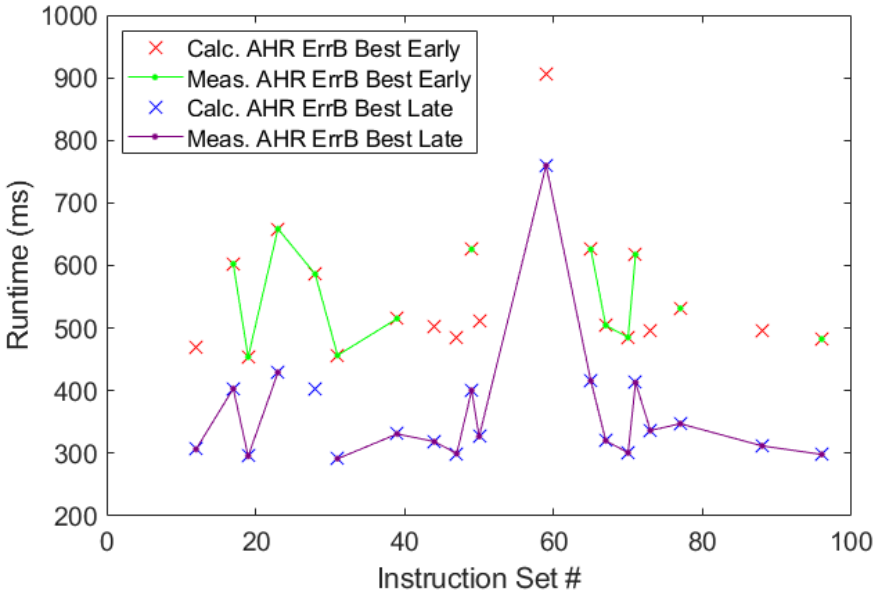

Figure 28. Calculated & Measured AHR TMR ErrB Best - Early & Late Runtimes

The average percent differences between the AHR TMR Type B Best - Early or Best - Late error program runtimes that were calculated and those measured were found using Equations 57 & 58.

$$PD_{Time\ AHR\ ErrBBE} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{AHR\ ErrBBE\ Meas} - T_{AHR\ ErrBBE\ Calc}}{T_{AHR\ ErrBBE\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{57}$$

$$PD_{Time\ AHR\ ErrBBL} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{AHR\ ErrBBL\ Meas} - T_{AHR\ ErrBBL\ Calc}}{T_{AHR\ ErrBBL\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{58}$$

The average percent difference for AHR Type B Best - Early is:

$$PD_{Time\ AHR\ ErrBBE} = 1.318 \cdot 10^{-3}\% \tag{59}$$

The average percent difference for AHR Type B Best - Late is:

$$PD_{Time\ AHR\ ErrBBL} = 1.441 \cdot 10^{-3}\% \tag{60}$$

The average percent differences indicate that the equations used to calculate the runtime performance of AHR MIPS with a TMR Type B Best - Early or Best - Late error closely predict the runtime that was measured in hardware. The average percent difference is non-zero due to signal propagation delays.

4.4.2.3 AHR TMR MIPS Type B Worst Errors

The runtime results for the 20 instruction sets processed by an AHR MIPS architecture experiencing a TMR Type B Worst - Early or TMR Type B Worst - Late error are shown on the same plot in Figure 29. Calculated results are shown for each of the 20 instruction sets for both the Type B Worst - Early & Late scenarios, but only 11 sets and 12 sets, respectively, had measured runtimes within $50\mu s$ of the calculated result and were considered valid. The invalid measurements were not collected because the measured runtimes were either faster than the calculated value or had a difference greater than $50\mu s$. These measurement-to-calculation runtime discrepancies were due to implementation issues that will be discussed in Section 4.5. Multiple calculated and measured data points overlap in Figure 29. To help distinguish between data points, the calculated AHR Type B Worst - Early values are marked by a red circle "o", and the measured AHR Type B Worst - Early values are marked by a green star. The differences between the Early or Late program runtimes that were calculated and those measured cannot be seen in this figure.

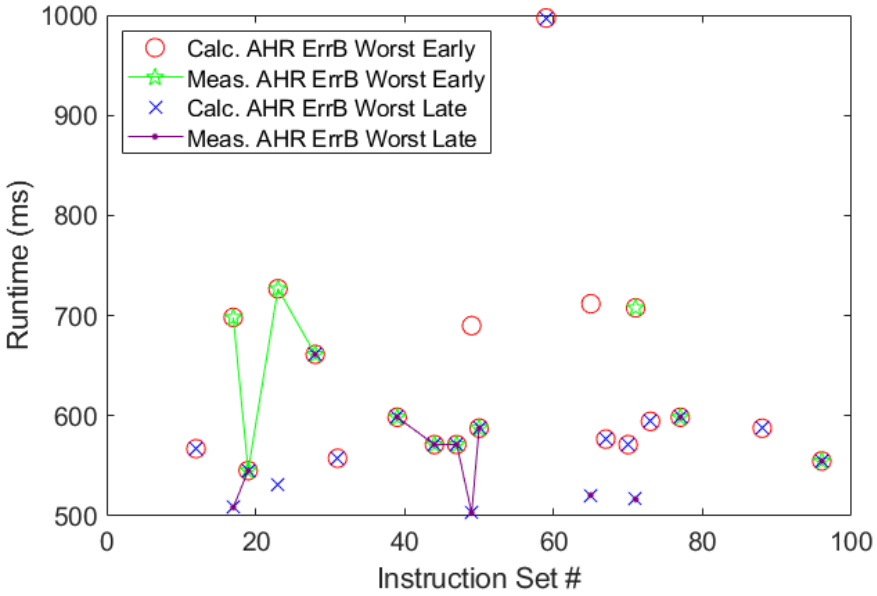

Figure 29. Calculated & Measured AHR TMR ErrB Worst - Early & Late Runtimes

The average percent differences between the AHR TMR Type B Worst - Early or Worst - Late error program runtimes that were calculated and those measured were found using Equations 61 & 62.

$$PD_{Time\ AHR\ ErrBWE} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{AHR\ ErrBWE\ Meas} - T_{AHR\ ErrBWE\ Calc}}{T_{AHR\ ErrBWE\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{61}$$

$$PD_{Time\ AHR\ ErrBWL} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{AHR\ ErrBWL\ Meas} - T_{AHR\ ErrBWL\ Calc}}{T_{AHR\ ErrBWL\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{62}$$

The average percent difference for AHR Type B Worst - Early is:

$$PD_{Time\ AHR\ ErrBWE} = 1.399 \cdot 10^{-3}\% \tag{63}$$

The average percent difference for AHR Type B Worst - Late is:

$$PD_{Time\ AHR\ ErrBWL} = 1.332 \cdot 10^{-3}\% \tag{64}$$

The average percent differences indicate that the equations used to calculate the runtime performance of AHR MIPS with a TMR Type B Worst - Early or Worst - Late error closely predict the runtime that was measured in hardware. The average percent difference is non-zero due to signal propagation delays.

4.4.2.4 AHR TSR MIPS Errors

The runtime results for the 20 instruction sets processed by an AHR MIPS architecture experiencing a TSR Early or Late error are shown on the same plot in Figure 30. Calculated results are shown for each of the 20 instruction sets for both the TSR Early & Late error scenarios, but only 17 sets and 12 sets, respectively, had measured runtimes within $50\mu s$ of the calculated result and were considered valid. The invalid measurements were not collected because the measured runtimes were either faster than the calculated value or had a difference greater than $50\mu s$. These measurement-to-calculation runtime discrepancies were due to implementation issues that will be discussed in Section 4.5. The differences between the Early or Late program runtimes that were calculated and those measured cannot be seen in this figure.

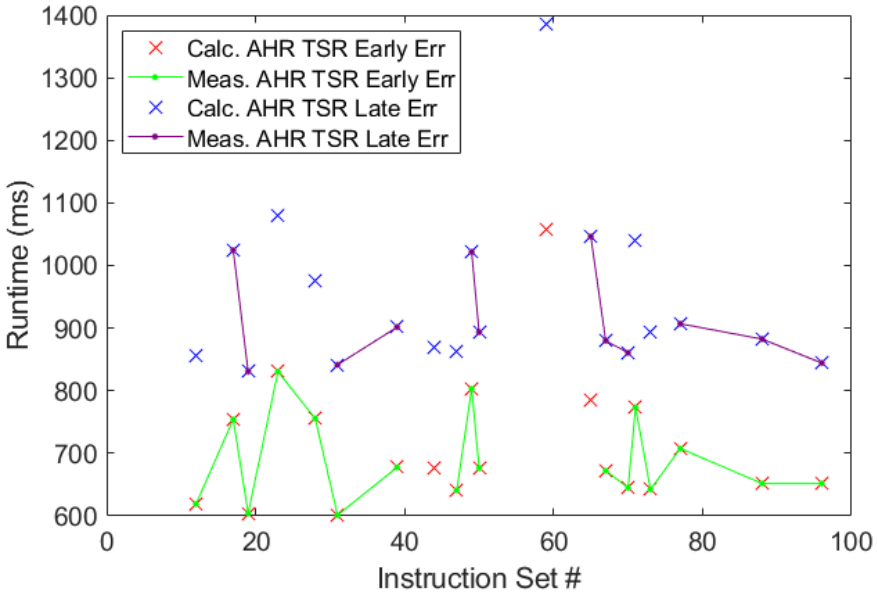

Figure 30. Calculated & Measured AHR TSR Early & Late Error Runtimes

The percent differences between the AHR TSR Early or Late error program runtimes that were calculated and those measured were found using Equations 65 & 66.

$$PD_{Time\ AHR\ TSR\ ErrE} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{AHR\ TSR\ ErrE\ Meas} - T_{AHR\ TSR\ ErrE\ Calc}}{T_{AHR\ TSR\ ErrE\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{65}$$

$$PD_{Time\ AHR\ TSR\ ErrL} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{AHR\ TSR\ ErrL\ Meas} - T_{AHR\ TSR\ ErrL\ Calc}}{T_{AHR\ TSR\ ErrL\ Calc}} \cdot 100\% \right]}{N_{programs}} \tag{66}$$

The average percent difference for AHR TSR Early error is:

$$PD_{Time\ AHR\ TSR\ ErrE} = 1.364 \cdot 10^{-3}\% \tag{67}$$

The average percent difference for AHR TSR Late error is:

$$PD_{Time\ AHR\ TSR\ ErrL} = 1.324 \cdot 10^{-3}\% \tag{68}$$

The average percent differences indicate that the equations used to calculate the runtime performance of AHR MIPS with a TSR Early or Late error closely predict the runtime that was measured in hardware. The average percent difference is non-zero due to signal propagation delays.

4.5 Hardware Measurement-to-Calculation Discrepancies

As mentioned throughout the discussion of results in Sections 4.2, 4.3, and 4.4, there were many hardware measurements that were not collected because the runtimes measured did not match the calculated runtimes. These mismatches indicated that either the timing equations were not properly predicting/calculating the hardware runtimes, or that the Xilinx Vivado (Version 2020.2) software used to program the Xilinx Spartan 7 FPGA development boards was incorrectly implementing the architecture in hardware.

The timing equations used to calculate program runtimes were provided by Hamilton in [20], modified for use in this research, and tested for correctness on a subset of the randomly selected instruction sets using simulations. The instruction subset was selected with the intent to exercise the extremes of the timing equations, to draw out edge cases or scenarios where the equations would not properly calculate the runtimes that were simulated. The equations were found to correctly predict the runtimes of the subset of instructions that were then verified to run in the same time in simulations. After the described verification was complete, there was high confidence going into hardware testing that the timing equations correctly calculated

the program runtimes for every operational mode possible for TMR, TSR, and AHR MIPS.

All EF architectures tested in hardware had program runtimes that matched the calculated runtimes. The measurement-to-calculation discrepancies only appeared on architectures where errors were injected. Out of the 320 total hardware measurements performed (20 instruction sets x 16 Architectures & Operating Modes), 43 measurements were rejected because the oscilloscope measurements of the runtime were either not within $50\mu s$ of the calculated/predicted value or were faster than the calculated/predicted value. At the end of data collection, only two instruction sets had a measurement-to-calculation match across all 16 different architectures and operating modes. Six instruction sets had one mismatch, four sets had two mismatches, and eight sets had three or more mismatches. The instruction sets with only one mismatch were prioritized for further analysis because correcting one mismatch took less work than the sets with more mismatches.

The six instruction sets with one mismatch were individually simulated and the runtime was compared against the calculated value. All sets showed simulation-to-calculation matches, which indicated that for those sets the calculated/predicted runtimes were correct. Thus, the measurement runtime discrepancy was hypothesized to have been caused by the way in which the architectures were implemented on the FPGA by the design software. To test this hypothesis, the hardware projects of Board #1 & #2 for each of the six instruction sets with one mismatch were completely rebuilt to determine if the mismatches were caused by a software generated implementation error. Hardware runtime measurements were made for the six re-generated instruction sets, and only 3/6 measurement-to-calculation discrepancies were corrected despite the simulations matching calculations for all six sets.

These results supported the hypothesis that the design decisions made by Vivado

during the hardware "Place and Route" steps were not correctly implementing some of the instruction sets. In most cases, when a mismatch occurred the runtime measured in hardware was faster than the predicted/calculated runtime. For nearly all AHR MIPS mismatches the measured runtimes were faster than calculated because that architecture was encountering unexpected errors that would cause it to operate in TMR mode for longer. Due to time constraints, the source of these errors was not able to be investigated further and in total only 5/20 instruction sets were able to be used for the analysis performed in Section 4.6.

4.6 AHR MIPS Performance Behavior Analysis

Calculated runtimes were obtained for all 20 instruction sets on each architecture for all operating modes. However, not all these runtimes were able to be verified/measured in hardware. This section will use the instruction sets that were able to be measured on each architecture for all operating modes to analyze the performance behaviors of AHR MIPS. These performance behaviors will be compared against the expected behaviors predicted in Ch. III Section 3.5.2.4 and the behaviors found through simulation in the previous AHR research [20].

Only 5/20 instruction sets were able to be measured across all operating modes for TMR, TSR, and AHR MIPS. The complete runtime results for the five instruction sets are shown in Figure 31. These instruction sets were sorted from the shortest to longest runtimes, as determined using the error free AHR MIPS runtimes. Sorting the data sets in this manner makes the performance behaviors easier to analyze and can provide insight into the ways certain errors affect the runtimes of programs with different lengths. Figure 31 will be referenced multiple times throughout this section, but groups of operational modes will be inspected in isolation to determine whether the measured results match predictions.

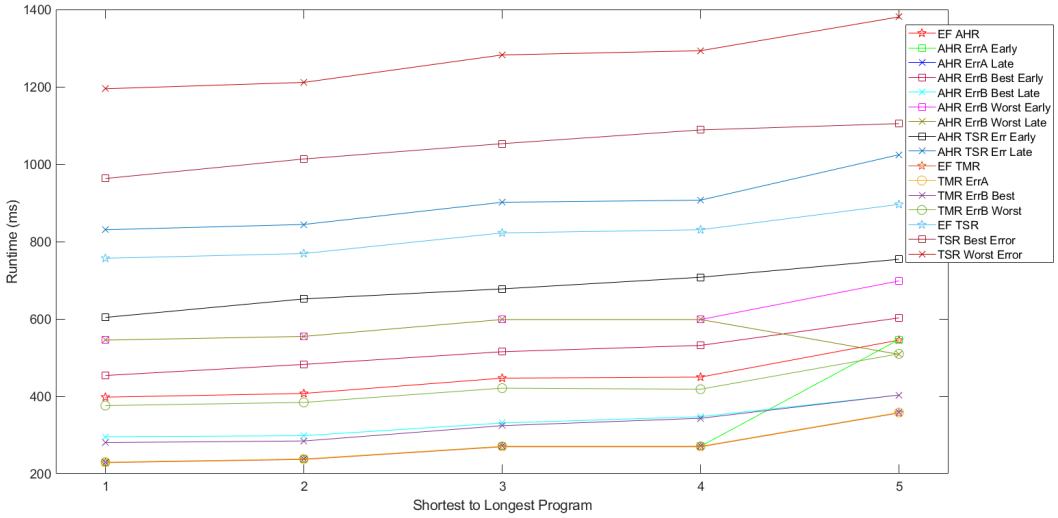

Figure 31. All Measured Runtimes of 5 Complete Instr. Sets (Sorted)

4.6.1 EF TMR, TSR, and AHR MIPS

The EF program runtimes of TMR, TSR, and AHR MIPS are shown separated from all other operational modes in Figure 32. EF TSR MIPS was predicted to run the slowest, EF TMR MIPS was predicted to run the fastest, and EF AHR MIPS was predicted to have runtimes that fell between TSR and TMR MIPS. The measured data presented in Figure 32 supports the predictions, and also mirrors the behaviors seen in the previous research [20].

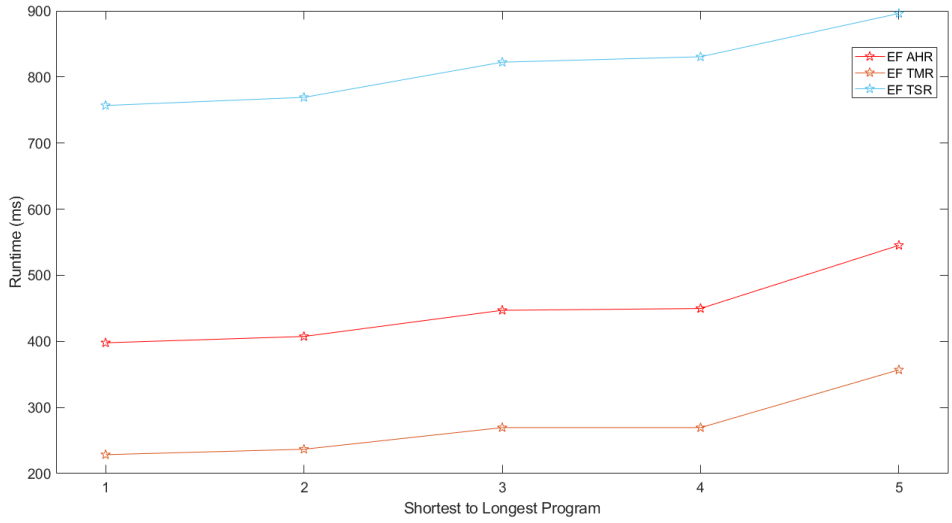

Figure 32. EF MIPS Measured Runtimes of 5 Complete Instr. Sets (Sorted)

Referencing Figure 31, EF TMR MIPS appears to have produced the fastest runtimes, but that conclusion cannot be made without further analysis.

4.6.2 TMR and AHR MIPS Type A Errors

The fastest runtimes appear to have been produced by EF TMR, AHR ErrA Early, AHR ErrA Late, and TMR ErrA, however the runtime differences between these operational modes cannot be seen in Figure 31. The only noticeable difference is that the AHR ErrA Early time for the last instruction set converges on the EF AHR runtime. Figure 33 focuses on the measured results of EF TMR, AHR ErrA Early, AHR ErrA Late, and TMR ErrA.

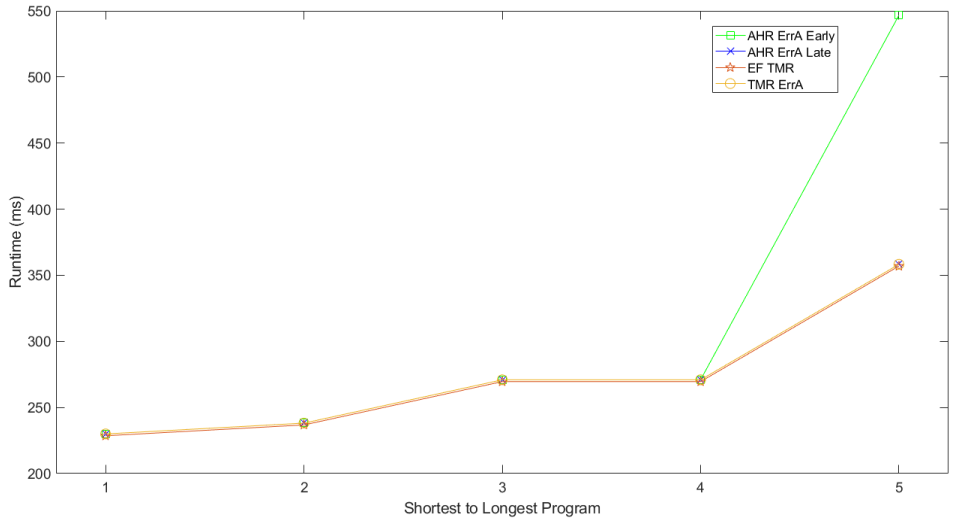

Figure 33. Focused view of Fastest Measured Runtimes of 5 Complete Instr. Sets (Sorted)

The fastest overall runtimes were produced by the EF TMR architecture which matches predictions, where EF TMR MIPS always processes programs the fastest. However, even with the focused view provided in Figure 33 the runtime differences between AHR ErrA Early, AHR ErrA Late, and TMR ErrA are indiscernible except for AHR ErrA Early on the last instruction set. AHR ErrA Early was noticeably different for the longest of the five instruction sets, because only Instruction Set #5 was long enough to allow AHR MIPS to transition to TSR mode, even after experiencing a Type A Early error. None of the other instruction sets were long enough to allow AHR MIPS to transition out of TMR mode. If AHR MIPS does not transition out of TMR mode, then it essentially operates as TMR MIPS with one extra component, the AHR Controller. This explains why the runtime performance of AHR ErrA Early, AHR ErrA Late, and TMR ErrA are so close that they are indiscernible. Despite being indiscernible, it was predicted that TMR ErrA would operate faster than both AHR ErrA Early & Late. This is because AHR MIPS signals must pass through the TMR Voter and the AHR Controller to reach memory,

whereas in TMR MIPS signals must only pass through the voter to reach memory, thus the path delay in AHR MIPS is always greater than TMR MIPS. To determine if this predicted behavior was realized in hardware requires additional analysis.

Upon further inspection and based upon the data shown in Figure 33, neither AHR ErrA Early, TMR ErrA, nor AHR ErrA Late establish steady runtime behaviors because no architecture was shown to always run faster than the others. For some of the instruction sets AHR ErrA Early or AHR ErrA Late ran faster than TMR ErrA. The runtime relationship between AHR ErrA Early or AHR ErrA Late and TMR ErrA does not match what was predicted. In calculations, excluding the last set, AHR ErrA Early, TMR ErrA, and AHR ErrA Late have the same runtime when AHR MIPS cannot transition to TSR mode. However, the calculations do not account for the path signals must take through real-world hardware components.

If the real-world signal propagation time through the AHR Controller could be considered zero, then TMR ErrA and AHR ErrA Early & Late should have equivalent runtimes, but the measured data sometimes shows AHR ErrA Early & Late operated slightly faster than TMR ErrA. Further analysis was performed by calculating the percentage difference between AHR Err A Early, AHR Err A Late, and TMR ErrA, shown in Table 7.

Table 7. Percent Difference Between TMR ErrA and both AHR ErrA Early & Late

Instr. Set	1	2	3	4	5
AHR ErrA Early	$3 \cdot 10^{-4}\%$	$-1 \cdot 10^{-5}\%$	$-3 \cdot 10^{-4}\%$	$-1 \cdot 10^{-4}\%$	52.57%
AHR ErrA Late	$1 \cdot 10^{-4}\%$	$5 \cdot 10^{-5}\%$	$-6 \cdot 10^{-4}\%$	$2 \cdot 10^{-5}\%$	$-3 \cdot 10^{-5}\%$

The percentage differences were found to be so small that the cause of this discrepancy between the predicted and measured behavior is most likely either due to

how Vivado chose to perform the Place & Route step for each architecture or caused by signal jitter in the DONE signal used to measure program runtimes in hardware. This explains the behaviors seen using the measured data for these operational modes, however the predicted behaviors of AHR Type A Early & Late are different when the programs are long enough to tolerate an error and can transition to TSR mode.

If the AHR MIPS program is long enough, then it was predicted that AHR ErrA Early would run slower than AHR ErrA Late, because a Type A Early error would not greatly affect the TMR to TSR transition point, and the resulting runtime would only be slightly more than EF AHR MIPS. A Type A Late error would push the transition point further into program execution causing AHR MIPS to operate longer in TMR mode, and the resulting runtime would be faster than EF AHR MIPS.

The predicted behavior of an AHR ErrA Early error was observed in hardware because for the longest instruction set in Figure 31, the measured runtime was slightly greater than the EF AHR MIPS runtime.

4.6.3 TMR and AHR MIPS Type B Errors

Figure 34 shows all of the Type B error operational modes for both TMR and AHR MIPS. The data for EF AHR and EF TMR are included to provide context into how the Type B errors for AHR and TMR MIPS affect program runtimes.

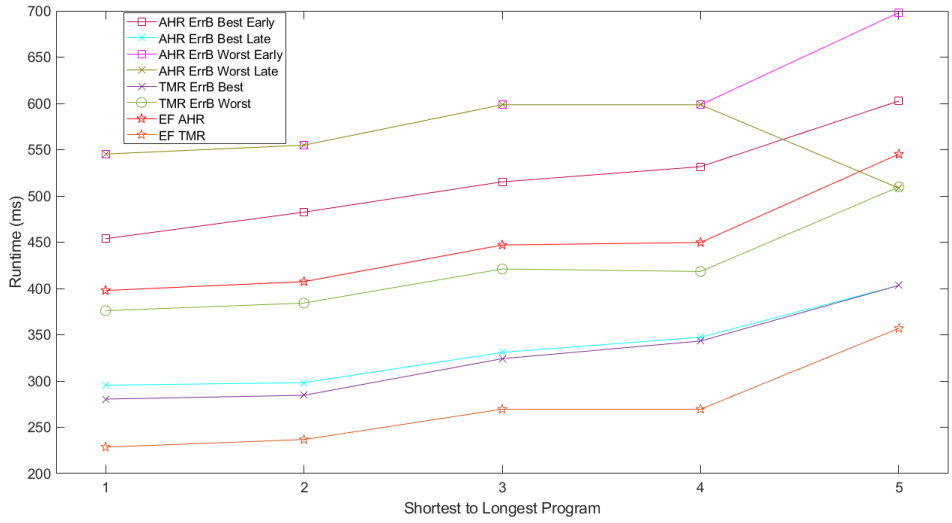

Figure 34. AHR & TMR MIPS Type B Errors Measured Runtimes of 5 Complete Instr. Sets (Sorted)

The TMR ErrB Worst error has a significant effect on the runtime of TMR programs, and produces runtimes that approach EF AHR, but do not ever equal EF AHR since EF AHR performs portions of the program in TSR mode.

In between the performance of TMR ErrB Worst and EF TMR are AHR ErrB Best - Late and TMR ErrB Best. TMR ErrB Best errors and AHR ErrB Best - Late errors appear to perform similarly, when the AHR Type B error is late enough to cause AHR MIPS to never transition to TSR mode. These results are interesting because for the longest instruction set, the runtime of TMR ErrB Best and AHR ErrB Best - Late are nearly identical, whereas the other points have noticeable performance differences. The AHR ErrB Best - Late errors are detected at the SW following SRP creation that contributes the least to the overall runtime, and TMR ErrB Best errors are detected in the same manner. This means that for the instruction sets presented in Figure 34, all AHR ErrB Best - Late runtimes should be nearly identical to TMR ErrB Best. Therefore, I believe there is an error in the equations I developed to both select the point at which to inject an AHR ErrB Best - Late error and calculate the

corresponding runtime. For four out of the five instruction sets, my calculations and corresponding hardware measurements appear to select the second-best location at which to inject an AHR ErrB Best - Late error and thus do not represent the true AHR ErrB Best - Late error scenario. Only, the longest instruction set showed the expected relationship between AHR ErrB Best late and TMR ErrB Best.

The AHR ErrB Worst - Early, AHR ErrB Worst - Late, and AHR ErrB Best - Early all produced runtimes slower than EF AHR MIPS. AHR ErrB Best - Early should run slower than EF AHR MIPS, because if the TMR to TSR transition point does not change due to the error, then the added runtime is equivalent to the time required to recover from the error and return to the point at which the error was detected from the loaded SRP.

Except for the last instruction set, the runtime differences between AHR ErrB Worst - Early & Late are indiscernible. This is an expected result because of how short the tested instruction sets were. For a short instruction set that can tolerate a Type B Worst class of error and still transition to TSR mode, there may only be one TMR SRP made and thus only one location to inject a Type B Worst error. For the first four instruction sets, there was only one location to inject a Type B error, which explains why the runtime differences are equivalent/indistinguishable. For the last instruction set, there were two locations to inject a Type B error. The AHR ErrB Worst - Early error did not affect the TMR to TSR transition point, but the Late error did. The late error caused one less loop to be performed in TSR mode, and thus the runtime of the last instruction set experiencing a Type B Worst - Late error was better than EF AHR MIPS, because more of the program was performed in TMR mode. The runtime of the last instruction set experiencing an AHR ErrB Worst - Late error converges on the TMR ErrB Worst runtime, which is an expected result in this case.

4.6.4 TSR and AHR MIPS Errors

Figure 35 shows all of the possible TSR error scenarios for both AHR and TSR MIPS. The data for EF AHR and EF TSR are included to provide context into how the various TSR errors affect program runtimes.

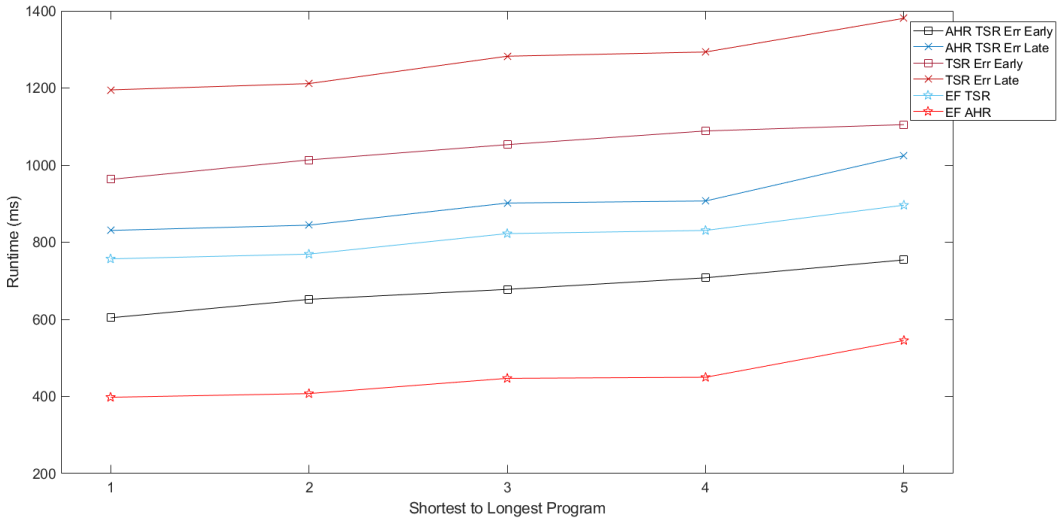

Figure 35. Measured Runtimes of Early & Late Errors for AHR and TSR MIPS of 5 Complete Instr. Sets (Sorted)

The longest runtimes measured were produced by the TSR MIPS architecture. Specifically, the longest times were caused by a TSR Late error and the second longest caused by TSR Early error. These measurements match predictions because the TSR instruction sets are longer than equivalent TMR sets, and as a result take significantly longer to process. Also, AHR MIPS always processes a portion of the program in TMR mode using the TMR instructions, thus AHR never processes an entire program in TSR mode and was found to never operate as slowly as either of the TSR error scenarios.

The AHR TSR Early error runtimes were slower than EF AHR MIPS and faster than EF TSR MIPS. This result matches the expected behavior because an Early

TSR error is detected by the first BNE-SW pair of the TSR portion of the AHR instruction set. This type of error is corrected by reloading an SRP, and re-processing instructions up-to the point at which the error was detected. Therefore, not enough TSR instructions are performed to make the AHR TSR Early error runtimes greater than EF TSR, but the resulting runtimes do move towards the EF TSR runtimes. On the other hand, an AHR TSR Late error is detected while writing an SRP at the end of a TSR loop. This error requires that error recovery occur and that an entire TSR loop be re-processed. The addition of an extra TSR loop causes the AHR TSR Late error runtimes to not only be slower than AHR TSR Early but also EF TSR MIPS.

4.6.5 Summary of AHR MIPS Performance Behavior Analysis

Figure 36 is a copy of Figure 31 and shows the five instruction sets that were sorted and used to determine the performance characteristics of TMR, TSR, and AHR MIPS in every tested error scenario.

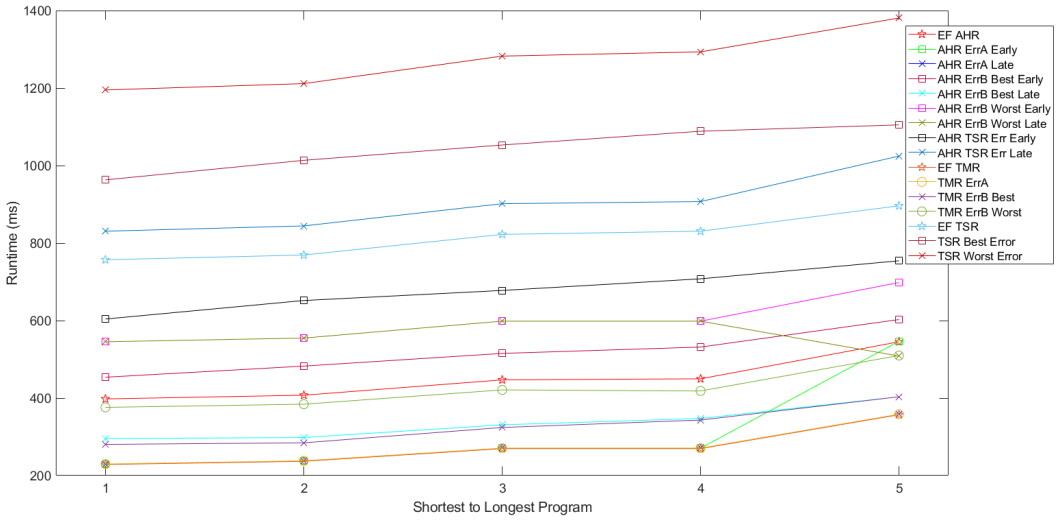

Figure 36. All Measured Runtimes of 5 Complete Instr. Sets (Sorted)

The previous research simulated much larger programs that performed 999 loops; therefore the observed performance behaviors and relationships are slightly different than what was observed in this research. However, nearly all the performance behaviors and relationships between the MIPS architectures physically measured were found to match calculations and the predictions made in Ch. III Section 3.5.2.4, despite using the smaller programs that were tested in this research. The only result that was different than what was expected for these smaller programs was the performance relationship between AHR ErrB Best - Late and TMR ErrB Best, which was discussed in Section 4.6.3. These hardware results for five complete instruction sets provide insight into how AHR MIPS operates when implemented on a COTS FPGA. However, further experimentation and developments will be necessary to overcome the implementation issues that did not allow for all 20 instruction sets to be used during behavioral analysis. A discussion of the future work for AHR MIPS research is given in Ch. V Section 5.2.

4.7 Summary of Results

Sections 4.2, 4.3, and 4.4 discussed the hardware results collected for TMR, TSR, and AHR MIPS operating under every possible error scenario. The results discussed in those sections highlighted how the timing equations used to calculate the runtime of a given program were able to accurately predict the runtime of that program measured in hardware. It was common to see percent differences of 10^{-3}%, when both a calculated and measured result existed. Section 4.5 discussed the challenges encountered during hardware measurements, sought to provide an explanation for those challenges, and indicated future work that can be conducted to deal with those challenges. Section 4.6 used the five instruction sets that had complete measurement-to-calculation matches to evaluate and compare the runtime performance of the newly developed

AHR MIPS against both TMR and TSR MIPS. The results demonstrated that AHR MIPS has flexible runtime performance characteristics, where it can perform nearly as well as error free TMR, never perform as poorly as TSR with errors, and can provide performance profiles in between those two extremes. The hardware performance profiles of AHR MIPS mostly match what was predicted, and have demonstrated the value AHR MIPS could offer space mission designers seeking a cost effective method of integrating advanced radiation mitigation techniques into a space system.

V. Conclusion

The conclusion of the research presented in this dissertation returns to a discussion of the research questions presented in Ch. I Section 1.2.

(a) Can AHR be implemented on a COTS FPGA(s)?

(b) What are the performance characteristics of AHR on a COTS FPGA? (Processor Performance)

(c) How does the measured performance of AHR compare to predictions and results produced in prior research?

Answering the first research question was the primary purpose of this research and the work presented throughout this dissertation showed that Adaptive-Hybrid Redundancy (AHR) applied to a Microprocessor Without Interlocked Pipeline Stages (MIPS) architecture can be successfully implemented on a Consumer Off-The-Shelf (COTS) Field-Programmable Gate Array (FPGA).

Ch. II Section 2.6 described the previous work performed by Hamilton to design and simulate AHR MIPS. During the previous research, two different attempts were made to implement AHR MIPS in FPGA hardware and neither were successful. The first attempt used a single FPGA board to implement AHR, but due to size constraints and the need to collect accurate power measurements, the single board implementation of AHR MIPS was not pursued further. The second attempt used two FPGA boards to split the AHR MIPS design, thus avoiding size constraint limitations and enabling more accurate power measurements of the AHR MIPS processor. However, this second attempt was not successful due to signal timing issues caused by the parallel communication scheme used for inter-board communication.

Ch. III of this dissertation addressed the previous AHR MIPS implementation issues by modifying the design to use a Universal Asynchronous Receiver-Transmitter (UART) based serial communication scheme for inter-board communication. Four communication modules were designed and added to the AHR MIPS architecture that enabled the architecture to successfully operate on a COTS FPGA. The COTS FPGA selected was the Xilinx Spartan 7 FPGA packaged in the Spartan 7 SP701 Evaluation Kit. By successfully implementing AHR MIPS in hardware using two Spartan 7 FPGAs the first research question can be answered as "Yes".

The second and third research questions were answered in Ch. IV, where the runtime performance of AHR MIPS was collected and then compared against collected results of Triple Modular Redundancy (TMR) MIPS and Temporal Software Redundancy (TSR) MIPS.

The runtime performance of AHR MIPS was determined by measuring the time it took to process programs. Program runtime measurements were collected while AHR MIPS operated without error and while AHR MIPS was subjected to simulated radiation induced errors. Either a single error caused by a Single Event Upset (SEU) or two errors caused by two separate SEUs were simulated. Various AHR MIPS error scenarios were produced depending on the point at which the error(s) were injected. Every operational mode (error free & w/error) of AHR MIPS was tested in hardware using 20 randomly generated programs. The runtime of each program processed by AHR MIPS for all operational modes was measured and enabled the performance of AHR MIPS on a COTS FPGA to be quantified, thus answering the second research question.

The runtime performance of TMR and TSR MIPS were also determined in the same way as described for AHR MIPS. Both TMR and TSR MIPS were injected with errors to determine the runtime of programs under similar error scenarios tested on

AHR MIPS. The same 20 randomly generated programs tested on AHR MIPS were used to test TMR and TSR MIPS. After collecting all runtime data for every program on every architecture, the measured performance of AHR MIPS was compared against TMR and TSR MIPS. The analysis performed in Ch. IV found that in all but one operational mode AHR MIPS performed in hardware as predicted during this research and by the prior research. The results found that AHR MIPS had flexible runtime performance, where at best AHR MIPS processed programs nearly as fast as Error Free (EF) TMR MIPS, and at worst processed programs slower than EF TSR MIPS. The performance analysis performed in Ch. IV Section 4.6 answered the third research question.

5.1 Contributions

Adaptive Hybrid Redundancy (AHR) was shown to correctly operate in Consumer Off-The-Shelf (COTS) Field Programmable Gate Array (FPGA) hardware and demonstrated nearly the same runtime behaviors that were seen in the previous research where AHR was simulated. AHR demonstrated correct operation during Triple Modular Redundancy (TMR) mode, Temporal Software Redundancy (TSR) mode, and while transitioning from TMR to TSR mode. Successful implementation of AHR in COTS FPGA hardware required the design of new hardware modules.

Four new communication modules specific to the Basic MIPS processors and memory used in AHR were designed, written in Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL), and integrated into the AHR architecture to facilitate communication between the Basic MIPS processor(s) on one FPGA board to the associated memory module on another FPGA board. The design of both the processor board and memory board, Board #1 and Board #2 respectively, were written in VHDL and used to implement AHR MIPS in hardware. The error

inject module used to perform fault injection was re-designed in VHDL to produce expected behavior when an error needed to be injected at an instruction that would also trigger the creation of a TMR Save Restore Point (SRP). The AHR Controller component of the AHR MIPS architecture was modified in VHDL to restart the architecture after a program completed which enabled consecutive program runs to be functionally equivalent. This modification was necessary to produce repeatable behavior that could be measured using an oscilloscope. A component named the LED Oscillator was designed, written in VHDL, and implemented to produce a reference DONE signal used by the oscilloscope to measure the runtimes of programs processed by the AHR MIPS architecture.

A Xilinx Vivado software specific workflow was developed to program each Spartan 7 SP701 FPGA board. This workflow used Vivado to package VHDL files as Custom Intellectual Property (IP) Blocks and integrate them into a block diagram used to define the hardware design that was transformed into a bitstream. The workflow also made use of Tool Command Language (TCL) scripting to remove the possibility of user error while defining the connections between blocks in the block diagram.

The timing equations/processes developed and written in MATLAB by Hamilton in [20] were modified to correctly calculate the runtimes of the short programs that were used for testing in this research. The previous research calculated the runtimes of much larger programs, and because of this consecutive program runs could not be observed. In this research short programs were used and because consecutive program runs could be observed, errors within the timing equations could be easily identified and corrected. The timing equations were also improved by adding additional equations/processes in MATLAB to account for new edge cases encountered during this research.

5.2 Future Work

While performing hardware measurements numerous program runtimes were not collected due to hypothesized implementation errors introduced by Vivado during the "Place and Route" step of design implementation. At the end of data collection, only 5/20 instruction sets had valid calculated and measured runtimes across all architectures and operational modes. The final performance/behavioral analysis of AHR MIPS using those five instruction sets was not as comprehensive due to the implementation errors that required the other 15 sets be excluded. Immediate future work would involve further investigation into the cause of the observed hardware implementation errors. If the current hypothesis is correct, then the way Vivado performs "Place and Route" will need to be researched in order to determine how to ensure every implemented design is stable and performs as simulated. Solving these implementation issues would enable a more thorough analysis of AHR MIPS in hardware to occur using all 20 instruction sets.

Immediate future work would be to investigate the discrepancy seen for the predicted relationship between AHR ErrB Best - Late and TMR ErrB Best in hardware. As discussed in Ch. IV Section 4.6.3, there may be an error with how the minimum runtime and associated error injection point and loop count are being selected to realize the true AHR ErrB Best - Late error scenario for every program in hardware.

After solving the implementation issues and correcting calculation discrepancies, future work on AHR MIPS should increase the baud rate at which the UART modules within the communications network operate. This research set the clock frequency of all MIPS processors to 50 MHz and used a UART baud rate of $115,200$. Although, this research was focused on getting AHR MIPS to function in hardware, the tested AHR MIPS architecture took a substantial runtime performance hit by using serial communication between the processors and memory. The UART-induced

delays dominated the calculated and measured runtimes of AHR MIPS programs and should be reduced. In future work, the baud rate should be increased from $115,200$ until communication errors begin to occur, at which point the highest baud rate that enables successful communication should be used. Increasing the UART baud rate would allow for longer instruction sets to be tested on the AHR MIPS architecture, which would only further mature the AHR MIPS technology towards use on actual systems that would require the execution of long programs.

The research presented in this dissertation only determined the performance of AHR MIPS using program runtimes. In the previous work by Hamilton [20], simulated results were presented for the energy usage of AHR MIPS and were compared against TMR and TSR MIPS. The energy usage was calculated using separate energy equations and were calculated with respect to a different FPGA chip than the Xilinx Spartan 7 used in this research. Since this research used Xilinx products, future work could use the power estimation tools available for Xilinx products to either modify the previously developed energy equations or create new ones. The Vivado software performs power analysis as part of the "Implementation" step in the hardware design workflow and returns estimates of on-chip power. Future work could use Vivado to perform an in-depth power analysis of just the FPGA that contains the MIPS processors used to implement TMR, TSR, or AHR MIPS and new energy equations could be made to predict the energy consumption of each in hardware. The hardware energy consumption could be measured using Spartan 7 FPGA boards that are only packaged with the chip and a few I/O interfaces to accurately measure only the energy used by the processors. Alternatively, there may be other Spartan 7 FPGA boards that either have or are compatible with, specialized power supplies that only power the FPGA chip, can measure the power utilization, and can report that power utilization to the user. If it is determined that there are no Spartan 7 FPGA boards

that enable accurate power measurement, the new AHR MIPS design could be implemented on another Xilinx FPGA that does offer the desired functionality. The AHR MIPS design is completely defined in VHDL and can be implemented on any two FPGA devices with enough resources to support the architecture.

Future work to enhance AHR MIPS would be to design and integrate a third operating mode into the architecture. Dual Modular Redundancy (DMR), described in Ch. II Section 2.4.1.2, could be added to the AHR MIPS architecture and would enhance the architecture's performance and radiation mitigation capabilities. For example, if AHR MIPS operating in TMR mode encounters a permanent SEE that damages one of the redundant modules, that module could continuously produce incorrect results for the remainder of that device's operational lifetime. A DMR mode could be introduced where the AHR Controller monitors the number of times a redundant module produces an error, and if it is above a threshold that redundant module is considered to be permanently damaged and is held in reset. The AHR Controller would then transition AHR MIPS to DMR mode, which would use the two redundant modules identified as undamaged to continue processing instructions. This mode would have reduced radiation tolerance compared to TMR, but could act as a backup tolerance mode during the aforementioned error scenario. DMR mode could also be used as an intermediate performance mode that does not use as much energy as TMR yet offers the same program runtime performance.

Improvements could also be made to the method of software redundancy used in AHR MIPS as future work. EDDI TSR was chosen as the software redundancy method for AHR MIPS, but as discussed in Ch. II Section 2.4.2.1 improved versions of EDDI TSR have been designed. One of these advanced EDDI methods could be integrated into the AHR MIPS architecture to increase the radiation tolerance of AHR MIPS while operating in TSR mode. Also, an entirely different method of

software redundancy could be integrated into the AHR MIPS architecture such as signature analysis. Signature analysis was discussed in Ch. II Section 2.4.2.2, and although these methods incur code size overheads like EDDI TSR they could provide enhanced radiation tolerance over any EDDI-like method of software redundancy.

Another important development necessary for AHR MIPS would be the inclusion of configuration memory scrubbing or dynamic partial reconfiguration to correct errors that accumulate in the configuration memory of the target FPGA. The AHR MIPS architecture developed in this research cannot correct radiation-induced configuration memory errors. Configuration memory scrubbing could be implemented using an off-chip radiation-hardened memory module to periodically re-write the configuration memory of AHR MIPS and could provide passive correction of configuration memory. Dynamic partial reconfiguration could also correct configuration memory, but would require that suspected configuration memory errors be identified, so only that portion of the FPGA would be reconfigured. The inclusion of either method into the AHR MIPS design would be necessary if AHR were to be implemented on an FPGA for use in a space system. FPGA designs intended for use in space must include redundancy for configuration memory and memory cells that hold user data. Configuration memory scrubbing and dynamic partial reconfiguration were described in Ch. II Sections 2.4.1.4 & 2.4.1.3 respectively.

The AHR MIPS architecture developed during this research has the processor and memory split across two FPGA boards. This design decision is one that would enable future work radiation testing the AHR MIPS architecture at a radiation test facility. Since the AHR MIPS processors are split from the memory, it is feasible to just use longer wires between Board #1 and #2 to facilitate radiation testing on Board #1. Longer wires could be used to move Board #2 either outside the radiation test chamber or some distance away from the radiation source where it

can be shielded. Physical radiation testing of AHR MIPS would be a necessary and important step towards determining the real-world feasibility of using AHR MIPS over a standalone TMR or TSR MIPS implementation. The error injection results collected during this research and any future error injection results would also require that real-world radiation test data be collected, before a space mission planner/designer would consider using AHR MIPS.

Appendix A. Detailed Design Tables

Table 8. AHR MIPS Instr. Subset

Instr.	Instr. Abbreviation	Instr. Name
1	SLL	Shift Word Left Logical
2	NOP	No operation
3	SRL	Shift Word Right Logical
4	SRA	Shift Word Right Arithmetic
5	SLLV	Shift Word Left Logical Variable
6	SRLV	Shift Word Right Logical Variable
7	SRAV	Shift Word Right Arithmetic Variable
8	LW	Load Word
9	SW	Store Word
10	ADD	Add Word
11	ADDU	Add Unsigned Word
12	SUB	Subtract Word
13	SUBU	Subtract Word Unsigned
14	AND	And
15	OR	Or
16	XOR	Exclusive Or
17	NOR	Nor
18	SLT	Set Less Than
19	SLTU	Set Less Than Unsigned
20	BGEZ	Branch on Greater Than or Equal to Zero
21	BLTZ	Branch on Less Than Zero

Table 8 – *Continued on next page*

Table 8 – *Continued from previous page*

Instr.	**Instr. Abbreviation**	**Instr. Name**
22	BEQ	Branch on Equal
23	BNE	Branch on Not Equal
24	BLEZ	Branch on Less Than or Equal to Zero
25	BGTZ	Branch on Greater Than Zero
26	ADDI	Add Immediate Word
27	ADDIU	Add Immediate Unsigned Word
28	SLTI	Set on Less Than Immediate
29	SLTIU	Set on Less Than Immediate Unsigned
30	ANDI	And Immediate
31	ORI	Or Immediate
32	XORI	Exclusive Or Immediate
33	LUI	Load Upper Immediate

Table 9. Architecture Specific Processes

MIPS Architecture	Process
TMR	SRP Creation
	Detect a Type A Error
	Perform Type A Error Recovery
	Return to Beginning of Instruction where Type A Error was Detected
	Detect a Type B Error
	Perform Type B Error Recovery
TSR	SRP 0 Creation
	SRP 1 Creation
	Error Recovery
AHR	TMR to TSR Transition Time

Table 10. MIBO FSM Read Operation States

Current State	Next State	Transition Condition	Description
s_idle	s_read0	i_READ = 1	Receive read signal from MIPS
s_read0	s_read1	i_DONE_TX = 1	Send read code and signal UART-TX
s_read1	s_read2	i_DONE_TX = 0	Wait for UART-TX to return to idle state
s_read2	s_read3	i_DONE_TX = 1	Send 1st byte of Address
s_read3	s_read4	i_DONE_TX = 0	Wait for UART-TX to return to idle state
s_read4	s_read5	i_DONE_TX = 1	Send 2nd byte of Address
s_read5	s_read6	i_DONE_TX = 0	Wait for UART-TX to return to idle state
s_read6	s_read7	i_READY = 1	Receive 1st byte of Address back from MEBI
s_read7	s_read8	i_READY = 0	Advance UART-RX towards idle state
s_read8	s_read9	i_READY = 1	Receive 2nd byte of Address back from MEBI Return UART-RX to idle state
s_read9	s_read10	i_READY = 0	Advance UART-RX towards idle state
s_read10	s_read11	i_C_DATA(15:0) == o_DATA(15:0)	Address data sent is equivalent to address data received Return UART-RX to idle state
s_read10	s_read10a	i_C_DATA(15:0) != o_DATA(15:0)	Address data sent is NOT equivalent to addr. data received Return UART-RX to idle state
s_read10a	s_read10b	i_DONE_TX = 1	Transmit 8-bit error code to MEBI
s_read10b	s_read2	i_DONE_TX = 0	Jump to s_read2 to re-do address transmit Wait for UART-TX to return to idle state
s_read11	s_read12	i_DONE_TX = 1	Transmit 8-bit correct code to MEBI
s_read12	s_read13	i_DONE_TX = 0	Wait for UART-TX to return to idle state
s_read13	s_idle	i_READ = 0	Return to idle state once MIPS lowers read signal

Table 11. MIBO FSM Write Operation States

Current State	**Next State**	**Transition Condition**	**Description**
s_idle	s_write0	i_WRITE = 1	Receive write signal from MIPS
s_write0	s_write1	i_DONE_TX = 1	Send write code and signal UART-TX
s_write1	s_write2	i_DONE_TX = 0	Return UART-TX to idle state
s_write2	s_write3	i_DONE_TX = 1	Send 1st byte of Address
s_write3	s_write4	i_DONE_TX = 0	Return UART-TX to idle state
s_write4	s_write5	i_DONE_TX = 1	Send 2nd byte of Address
s_write5	s_write6	i_DONE_TX = 0	Return UART-TX to idle state
s_write6	s_write7	i_READY = 1	Receive 1st byte of Address back from MEBI
s_write7	s_write8	i_READY = 0	Advance UART-RX towards idle state
s_write8	s_write9	i_READY = 1	Receive 2nd byte of Address back from MEBI Return UART-RX to idle state
s_write9	s_write10	i_READY = 0	Advance UART-RX towards idle state

Table 11 – *Continued on next page*

Table 11 – *Continued from previous page*

Current State	**Next State**	**Transition Condition**	**Description**
s_write10	s_write11	i_C_DATA(15:0) == o_DATA(15:0)	Address data sent is equivalent to address data received Return UART-RX to idle state
s_write10	s_write10a	i_C_DATA(15:0) != o_DATA(15:0)	Address data sent is NOT equivalent to address data received Return UART-RX to idle state
s_write10a	s_write10b	i_DONE_TX = 1	Transmit 8-bit error code to MEBI and signal UART-TX
s_write10b	s_write2	i_DONE_TX = 0	Jump to s_write2 to re-do address transmit Return UART-TX to idle state
s_write11	s_write12	i_DONE_TX = 1	Transmit 8-bit correct code to MEBI
s_write12	s_write13	i_DONE_TX = 0	Return UART-TX to idle state
s_write13	s_write14	i_DONE_TX = 1	Send 1st byte of data
s_write14	s_write15	i_DONE_TX = 0	Return UART-TX to idle state
s_write15	s_write16	i_DONE_TX = 1	Send 2nd byte of data
s_write16	s_write17	i_DONE_TX = 0	Return UART-TX to idle state
s_write17	s_write18	i_DONE_TX = 1	Send 3rd byte of data

Table 11 – *Continued on next page*

Table 11 – *Continued from previous page*

Current State	**Next State**	**Transition Condition**	**Description**
s_write18	s_write19	i_DONE_TX = 0	Return UART-TX to idle state
s_write19	s_write20	i_DONE_TX = 1	Send 4th byte of data
s_write20	s_write21	i_DONE_TX = 0	Return UART-TX to idle state
s_write21	s_write22	i_READY = 1	Receive 1st byte of data from MEBI
s_write22	s_write23	i_READY = 0	Advance UART-RX towards idle state
s_write23	s_write24	i_READY = 1	Receive 2nd byte of data from MEBI
s_write24	s_write25	i_READY = 0	Advance UART-RX towards idle state
s_write25	s_write26	i_READY = 1	Receive 3rd byte of data from MEBI
s_write26	s_write27	i_READY = 0	Advance UART-RX towards idle state
s_write27	s_write28	i_READY = 1	Receive 4th byte of data from MEBI
s_write28	s_write29	i_READY = 0	Advance UART-RX towards idle state
s_write29	s_write30	i_C_DATA(31:0) == o_DATA(31:0)	Data sent is equivalent to data received Return UART-RX to idle state

Table 11 – *Continued on next page*

Table 11 – *Continued from previous page*

Current State	**Next State**	**Transition Condition**	**Description**
s_write29	s_write29a	i_C_DATA(31:0) != o_DATA(31:0)	Data sent is NOT equivalent to data received Return UART-RX to idle state
s_write29a	s_write29b	i_DONE_TX = 1	Transmit 8-bit error code to MEBI and signal UART-TX
s_write29b	s_write13	i_DONE_TX = 0	Jump to s_write13 to re-do data transmit Return UART-TX to idle state
s_write30	s_write31	i_DONE_TX = 1	Transmit 8-bit correct code to MEBI
s_write31	s_write32	i_DONE_TX = 0	Return UART-TX to idle state
s_write32	s_idle	i_WRITE = 0	Return to idle state once MIPS lowers write signal

Table 12. MEBI FSM Read Operation States

Current State	Next State	Transition Condition	Description
s_idle	s_read0	i_DATA = READ CODE i_READY = 1	Receive 8-bit read code from MIBO
s_read0	s_read1	i_READY = 0	Advance UART-RX towards idle state
s_read1	s_read2	i_READY = 1	Receive 1st byte of Address back from MIBO
s_read2	s_read3	i_READY = 0	Advance UART-RX towards idle state
s_read3	s_read4	i_READY = 1	Receive 2nd byte of Address back from MIBO
s_read4	s_read5	i_READY = 0	Advance UART-RX towards idle state
s_read5	s_read6	i_DONE_TX = 1	Transmit 1st byte of address to MIBO
s_read6	s_read7	i_DONE_TX = 0	Return UART-TX to idle state
s_read7	s_read8	i_DONE_TX = 1	Transmit 2nd byte of address to MIBO
s_read8	s_read9	i_DONE_TX = 0	Return UART-TX to idle state
s_read9	s_read10	i_DATA = CORRECT CODE i_READY = 1	Receive 8-bit correct code from MIBO
s_read9	s_read9a	i_DATA = ERROR CODE i_READY = 1	Receive 8-bit error code from MIBO
s_read9a	s_read9b	i_READY = 0	Advance UART-RX towards idle state
s_read9b	s_read1		Jump to s_read1 to re-receive address
s_read10	s_read11	i_READY = 0	Advance UART-RX towards idle state
s_read11	s_read12	i_INSTR_COMP_ACK = 1	Set memory read enable high and supply read address until instruction complete signal is set high
s_read12	s_idle	i_INSTR_COMP_ACK = 0	Clear output to memory once instruction complete signal is set low

Table 13. MEBI FSM Write Operation States

Current State	**Next State**	**Transition Condition**	**Description**
s_idle	s_write0	i_DATA = WRITE CODE i_READY = 1	Receive 8-bit write code from MIBO
s_write0	s_write1	i_READY = 0	Advance UART-RX towards idle state
s_write1	s_write2	i_READY = 1	Receive 1st byte of Address from MIBO
s_write2	s_write3	i_READY = 0	Advance UART-RX towards idle state
s_write3	s_write4	i_READY = 1	Receive 2nd byte of Address from MIBO
s_write4	s_write5	i_READY = 0	Advance UART-RX towards idle state
s_write5	s_write6	i_DONE_TX = 1	Transmit 1st byte of address to MIBO
s_write6	s_write7	i_DONE_TX = 0	Return UART-TX to idle state
s_write7	s_write8	i_DONE_TX = 1	Transmit 2nd byte of address to MIBO
s_write8	s_write9	i_DONE_TX = 0	Return UART-TX to idle state
s_write9	s_write10	i_DATA = CORRECT CODE i_READY = 1	Receive 8-bit correct code from MIBO

Table 13 – *Continued on next page*

Table 13 – *Continued from previous page*

Current State	**Next State**	**Transition Condition**	**Description**
s_write9	s_write9a	i_DATA = ERROR CODE i_READY = 1	Receive 8-bit error code from MIBO
s_write9a	s_write1	i_READY = 0	Advance UART-RX towards idle state Jump to s_write1 to re-receive address
s_write10	s_write11	i_READY = 0	Advance UART-RX towards idle state
s_write11	s_write12	i_READY = 1	Receive 1st byte of data from MIBO
s_write12	s_write13	i_READY = 0	Advance UART-RX towards idle state
s_write13	s_write14	i_READY = 1	Receive 2nd byte of data from MIBO
s_write14	s_write15	i_READY = 0	Advance UART-RX towards idle state
s_write15	s_write16	i_READY = 1	Receive 3rd byte of data from MIBO
s_write16	s_write17	i_READY = 0	Advance UART-RX towards idle state
s_write17	s_write18	i_READY = 1	Receive 4th byte of data from MIBO
s_write18	s_write19	i_READY = 0	Advance UART-RX towards idle state
s_write19	s_write20	i_DONE_TX = 1	Transmit 1st byte of data to MIBO

Table 13 – *Continued on next page*

Table 13 – *Continued from previous page*

Current State	**Next State**	**Transition Condition**	**Description**
s_write20	s_write21	i_DONE_TX = 0	Return UART-TX to idle state
s_write21	s_write22	i_DONE_TX = 1	Transmit 2nd byte of data to MIBO
s_write22	s_write23	i_DONE_TX = 0	Return UART-TX to idle state
s_write23	s_write24	i_DONE_TX = 1	Transmit 3rd byte of data to MIBO
s_write24	s_write25	i_DONE_TX = 0	Return UART-TX to idle state
s_write25	s_write26	i_DONE_TX = 1	Transmit 4th byte of data to MIBO
s_write26	s_write27	i_DONE_TX = 0	Return UART-TX to idle state
s_write27	s_write28	i_DATA = CORRECT CODE i_READY = 1	Receive 8-bit correct code from MIBO
s_write27	s_write27a	i_DATA = ERROR CODE i_READY = 1	Receive 8-bit error code from MIBO
s_write27a	s_write11	i_READY = 0	Advance UART-RX towards idle state Jump to s_write11 to re-receive address
s_write28	s_write29	i_READY = 0	Advance UART-RX towards idle state

Table 13 – *Continued on next page*

Table 13 – *Continued from previous page*

Current State	**Next State**	**Transition Condition**	**Description**
s_write29	s_write30	i_INSTR_COMP_ACK = 1	Set memory write enable high and supply address and write data until instruction complete signal is set high
s_write30	s_idle	i_INSTR_COMP_ACK = 0	Clear output to memory once instruction complete signal is set low

Table 14. MEBO FSM Read Operation States

Current State	**Next State**	**Transition Condition**	**Description**
s_idle	s_read0	(i_READ = 1 and i_MEM_READY = 1)	Receive read signal from MEBI and memory ready signal
s_read0	s_read1	i_DONE_TX = 1	Send 8-bit read code to MIBI
s_read1	s_read2	i_DONE_TX = 0	Return UART-TX to idle state
s_read2	s_read3	i_DONE_TX = 1	Send 1st byte of data
s_read3	s_read4	i_DONE_TX = 0	Return UART-TX to idle state
s_read4	s_read5	i_DONE_TX = 1	Send 2nd byte of data
s_read5	s_read6	i_DONE_TX = 0	Return UART-TX to idle state
s_read6	s_read7	i_DONE_TX = 1	Send 3rd byte of data
s_read7	s_read8	i_DONE_TX = 0	Return UART-TX to idle state
s_read8	s_read9	i_DONE_TX = 1	Send 4th byte of data
s_read9	s_read10	i_DONE_TX = 0	Return UART-TX to idle state
s_read10	s_read11	i_READY = 1	Receive 1st byte of data from MIBI
s_read11	s_read12	i_READY = 0	Advance UART-RX towards idle state

Table 14 – *Continued on next page*

Table 14 – *Continued from previous page*

Current State	**Next State**	**Transition Condition**	**Description**
s_read12	s_read13	i_READY = 1	Receive 2nd byte of data from MIBI
s_read13	s_read14	i_READY = 0	Advance UART-RX towards idle state
s_read14	s_read15	i_READY = 1	Receive 3rd byte of data from MIBI
s_read15	s_read16	i_READY = 0	Advance UART-RX towards idle state
s_read16	s_read17	i_READY = 1	Receive 4th byte of data from MIBI
s_read17	s_read18	i_READY = 0	Advance UART-RX towards idle state
s_read18	s_read19	i_C_DATA(31:0) == o_DATA(31:0)	Data sent is equivalent to data received Return UART-RX to idle state
s_read18	s_read18a	i_C_DATA(31:0) != o_DATA(31:0)	Data sent is NOT equivalent to data received Return UART-RX to idle state
s_read18a	s_read18b	i_DONE_TX = 1	Transmit 8-bit error code to MIBI
s_read18b	s_read2	i_DONE_TX = 0	Jump to s_read2 to re-do data transmit Return UART-TX to idle state
s_read19	s_read20	i_DONE_TX = 1	Transmit 8-bit correct code to MEBI

Table 14 – *Continued on next page*

Table 14 – *Continued from previous page*

Current State	**Next State**	**Transition Condition**	**Description**
s_read20	s_read21	i_DONE_TX = 0	Return UART-TX to idle state
s_read21	s_read22	i_MEM_READY = 0	Set instruction complete signal to MEBI high
s_read22	s_idle		Set instruction complete signal low and return to idle state

Table 15. MEBO FSM Write Operation States

Current State	Next State	Transition Condition	Description
s_idle	s_write0	(i_WRITE = 1 and i_MEM_READY = 1)	Receive write signal from MEBI and memory ready signal
s_write0	s_write1	i_DONE_TX = 1	Send 8-bit ready code to MIBI
s_write1	s_write2	i_DONE_TX = 0	Return UART-TX to idle state
s_write2	s_write3	i_MEM_READY = 0	Set instruction complete signal to MEBI high
s_write3	s_idle		Set instruction complete signal low and return to idle state

Table 16. MEBO FSM Program End Operation States

Current State	Next State	Transition Condition	Description
s_idle	s_reset0	i_PRGM_DONE = 1	Receive program done signal from memory
s_reset0	s_reset1	i_DONE_TX = 1	Send 8-bit program done code to MIBI and set instruction complete signal to MEBI high
s_reset1	s_reset2	i_DONE_TX = 0	Return UART-TX to idle state
s_reset2	s_idle		Set instruction complete signal low and return to idle state

Table 17. MIBI FSM Read Operation States

Current State	**Next State**	**Transition Condition**	**Description**
s_idle	s_read0	(i_READ = 1 and i_READY = 1)	Receive read signal from MIBO and ready signal from MEBO
s_read0	s_read1	i_READY = 0	Advance UART-RX towards idle state
s_read1	s_read2	i_READY = 1	Receive 1st byte of data from MEBO
s_read2	s_read3	i_READY = 0	Advance UART-RX towards idle state
s_read3	s_read4	i_READY = 1	Receive 2nd byte of data from MEBO
s_read4	s_read5	i_READY = 0	Advance UART-RX towards idle state
s_read5	s_read6	i_READY = 1	Receive 3rd byte of data from MEBO
s_read6	s_read7	i_READY = 0	Advance UART-RX towards idle state
s_read7	s_read8	i_READY = 1	Receive 4th byte of data from MEBO
s_read8	s_read9	i_READY = 0	Advance UART-RX towards idle state
s_read9	s_read10	i_DONE_TX = 1	Transmit 1st byte of data back to MEBO
s_read10	s_read11	i_DONE_TX = 0	Return UART-TX to idle state
s_read11	s_read12	i_DONE_TX = 1	Transmit 2nd byte of data back to MEBO

Table 17 – *Continued on next page*

Table 17 – *Continued from previous page*

Current State	**Next State**	**Transition Condition**	**Description**
s_read12	s_read13	i_DONE_TX = 0	Return UART-TX to idle state
s_read13	s_read14	i_DONE_TX = 1	Transmit 3rd byte of data back to MEBO
s_read14	s_read15	i_DONE_TX = 0	Return UART-TX to idle state
s_read15	s_read16	i_DONE_TX = 1	Transmit 4th byte of data back to MEBO
s_read16	s_read17	i_DONE_TX = 0	Return UART-TX to idle state
s_read17	s_read18	i_DATA = CORRECT CODE i_READY = 1	Receive 8-bit correct code from MEBO
s_read17	s_read17a	i_DATA = ERROR CODE i_READY = 1	Receive 8-bit error code from MEBO
s_read17a	s_read1	i_READY = 0	Jump to s_read1 to re-receive data Advance UART-RX towards idle state
s_read18	s_read19	i_READY = 0	Advance UART-RX towards idle state
s_read19	s_idle	i_READ = 0	Send read data to MIPS processor

Table 18. MIBI FSM Write Operation States

Current State	Next State	Transition Condition	Description
s_idle	s_write0	(i_WRITE = 1 and i_READY = 1)	Receive write signal from MIBO and ready signal from MEBO
s_write0	s_write1	i_READY = 0	Advance UART-RX towards idle state
s_write1	s_idle	i_WRITE = 0	Set ready signal to MIPS proc. high to begin execution of the next instruction

Table 19. MIBI FSM Program End Operation States

Current State	Next State	Transition Condition	Description
s_idle	s_reset0	i_DATA = RESET CODE	Receive reset code from MEBO
s_reset0	s_reset1	i_READY = 0	Advance UART-RX towards idle state
s_reset1	s_reset2		Set prgm done signal to MIPS proc. high to initiate reset
s_reset2	s_idle		Set prgm done signal low and return to idle state

Bibliography

1. P. Adell, G. Allen, G. Swift, and S. McClure, "Assessing and mitigating radiation effects in Xilinx SRAM FPGAs," *Proceedings of the European Conference on Radiation and its Effects on Components and Systems, RADECS*, pp. 418–424, 2008.

2. S. A. Aketi, J. Mekie, and H. Shah, "Single-Error Hardened and Multiple-Error Tolerant Guarded Dual Modular Redundancy Technique," *Proceedings of the IEEE International Conference on VLSI Design*, vol. 2018-January, pp. 250–255, 2018.

3. S. Bahramnejad and H. R. Zarandi, "An adaptive redundancy oriented method to tolerate soft errors in SRAM-based FPGAs using unused resources," *ARES 2010 - 5th International Conference on Availability, Reliability, and Security*, pp. 119–124, 2010.

4. R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–315, sep 2005.

5. J. Benfica, B. Green, B. C. Porcher, L. B. Poehls, F. Vargas, N. H. Medina, N. Added, V. A. De Aguiar, E. L. MacChione, F. Aguirre, and M. A. Da Silveira, "Analysis of FPGA SEU sensitivity to combined effects of conducted EMI and TID," *2016 Asia-Pacific International Symposium on Electromagnetic Compatibility, APEMC 2016*, vol. 63, no. 2, pp. 887–889, 2016.

6. M. Berg, C. Poivey, D. Petrick, D. Espinosa, A. Lesea, K. A. 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," in *IEEE Transactions on Nuclear Science*, vol. 55, no. 4, aug 2008, pp. 2259–2266.

7. M. D. Berg, H. S. Kim, A. D. Phan, C. M. Seidlick, K. A. Label, and J. A. Pellish, "Single event induced multiple bit errors and the effects of logic masking," *IEEE Transactions on Nuclear Science*, vol. 60, no. 6, pp. 4192–4199, 2013.

8. P. Bernardi, S. Member, L. Maria, V. Bolzani, S. Member, M. Rebaudengo, M. S. Reorda, F. L. Vargas, and M. Violante, "for Systems-on-a-Chip," vol. 55, no. 2, pp. 185–198, 2006.

9. L. Bozzoli, C. De Sio, B. Du, and L. Sterpone, "A Neutron Generator Testing Platform for the Radiation Analysis of SRAM-based FPGAs," *Conference Record*

- *IEEE Instrumentation and Measurement Technology Conference*, vol. 2021-May, pp. 1–5, 2021.

10. G. Bruguier and J. M. Palau, "Single Particle-Induced Late," *IEEE Transactions on Nuclear Science*, vol. 43, no. 2, pp. 522–532, 1996.

11. M. Caffrey, K. Morgan, D. Roussel-Dupre, S. Robinson, A. Nelson, A. Salazar, M. Wirthlin, W. Howes, and D. Richins, "On-orbit flight results from the reconfigurable cibola flight experiment satellite (CFESat)," in *Proceedings - IEEE Symposium on Field Programmable Custom Computing Machines, FCCM 2009*, 2009, pp. 3–10.

12. P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. S. Reorda, and M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors," in *IEEE Transactions on Nuclear Science*, vol. 47, no. 6 III, dec 2000, pp. 2231–2236.

13. S. L. Clark, K. Avery, and R. Parker, "TID and SEE testing results of altera cyclone Field Programmable Gate Array," *IEEE Radiation Effects Data Workshop*, pp. 88–90, 2004.

14. S. Cuenca-Asensi, A. Martínez-Álvarez, F. Restrepo-Calle, F. R. Palomo, H. Guzmán-Miranda, and M. A. Aguirre, "A novel co-design approach for soft errors mitigation in embedded systems," in *IEEE Transactions on Nuclear Science*, vol. 58, no. 3 PART 2, jun 2011, pp. 1059–1065.

15. T. A. Delong, B. W. Johnson, and J. A. Profeta, "A Fault Injection Technique for VHDL Behavioral-Level Models," *IEEE Design and Test of Computers*, pp. 24–33, 1996.

16. R. Glein, F. Rittner, and A. Heuberger, "Adaptive single-event effect mitigation for dependable processing systems," in *2016 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2016*. Institute of Electrical and Electronics Engineers Inc., 2016.

17. R. Glein, B. Schmidt, F. Rittner, J. Teich, and D. Ziener, "A self-adaptive SEU mitigation system for FPGAs with an internal block RAM radiation particle sensor," in *Proceedings - 2014 IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines, FCCM 2014*. Institute of Electrical and Electronics Engineers Inc., jul 2014, pp. 251–258.

18. O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-error detection using control flow assertions," *Proceedings - IEEE International Symposium*

on Defect and Fault Tolerance in VLSI Systems, vol. 2003-Janua, pp. 581–588, 2003.

19. N. Hamilton, S. Graham, T. Carbino, J. Petrosky, and A. Betances, "Adaptive-hybrid redundancy with error injection," *Electronics (Switzerland)*, vol. 8, no. 11, 2019.

20. N. S. Hamilton, "Adaptive-Hybrid Redundancy for Radiation Hardening," Ph.D. dissertation, Air Force Institute of Technology, March 2019.

21. ——, "Adaptive-Hybrid Redundancy MIPS Architecture Version 2.2," Jul 2019.

22. ——, "Basic MIPS Architecture Verison 1.4," Jul 2019.

23. ——, "Triple Modular Redundancy MIPS Architecture Version 1.4," Jul 2019.

24. R. W. Hamming, "The Bell system technical journal," *The Bell System Technical Journal*, vol. 196, no. 2, pp. 147–160, 1950.

25. P. Ipa and S. Di, "DE10-Standard User Manual," Terasic Inc., Tech. Rep., 2017. [Online]. Available: https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-5505271707235-de10-standard-user-manual-sm.pdf

26. A. Jacobs, G. Cieslewski, A. D. George, A. Gordon-Ross, and H. Lam, "Reconfigurable fault tolerance: A comprehensive framework for reliable and adaptive FPGA-based space computing," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 5, no. 4, pp. 1–30, dec 2012. [Online]. Available: https://dl.acm.org/doi/10.1145/2392616.2392619

27. S. Kasap, E. W. Wachter, X. Zhai, S. Ehsan, and K. McDonald-Maier, "Survey of Soft Error Mitigation Techniques Applied to LEON3 Soft Processors on SRAM-Based FPGAs," *IEEE Access*, vol. 8, pp. 28 646–28 658, 2020.

28. F. Kastensmidt and P. Rech, *FPGAs and parallel architectures for aerospace applications: Soft errors and fault-tolerant design*, 2015. [Online]. Available: https://link.springer.com/content/pdf/10.1007/978-3-319-14352-1.pdf

29. R. Koga, S. H. Penzin, K. B. Crawford, and W. R. Crain, "Single event functional interrupt (SEFI) sensitivity in microcircuits," *Proceedings of the European Conference on Radiation and its Effects on Components and Systems, RADECS*, pp. 311–318, 1998.

30. C. A. Mao, Y. Xie, Y. Xie, H. Chen, and H. Shi, "An Automated Fault Injection Platform for Fault Tolerant FFT Implemented in SRAM-Based FPGA," *International System on Chip Conference*, vol. 2018-Septe, pp. 7–12, 2019.

31. Mips Technologies, "MIPS ® Architecture For Programmers Volume II-A : The MIPS32 ® Instruction Set," *Architecture*, vol. II, pp. 2008–2010, 2010.

32. K. S. Morgan, D. L. McMurtrey, B. H. Pratt, and M. J. Wirthlin, "A comparison of TMR with alternative fault-tolerant design techniques for FPGAs," in *IEEE Transactions on Nuclear Science*, vol. 54, no. 6, dec 2007, pp. 2065–2072.

33. H. T. Nguyen and Y. Yagil, "A systematic approach to SER estimation and solutions," *IEEE International Reliability Physics Symposium Proceedings*, vol. 2003-January, pp. 60–70, 2003.

34. B. Nicolescu, Y. Savaria, S. Member, and R. Velazco, "Coverage Against Single Bit-Flip Faults," *IEEE Transactions on Nuclear Science*, vol. 51, no. 6, pp. 3510–3518, 2004.

35. N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

36. ——, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

37. P. S. Ostler, M. P. Caffrey, D. S. Gibelyou, P. S. Graham, K. S. Morgan, B. H. Pratt, H. M. Quinn, and M. J. Wirthlin, "SRAM FPGA reliability analysis for harsh radiation environments," in *IEEE Transactions on Nuclear Science*, vol. 56, no. 6, dec 2009, pp. 3519–3526.

38. G. Popov, M. Nenova, and K. Raynova, "Reliability Investigation of TMR and DMR Systems with Global and Partial Reservation," *2018 7th Balkan Conference on Lighting, BalkanLight 2018 - Proceedings*, pp. 13–16, 2018.

39. B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin, "Improving FPGA Design Robustness with Partial TMR," Tech. Rep. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4017162/

40. G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," *Proceedings of the 2005 International Symposium on Code Generation and Optimization, CGO 2005*, vol. 2005, pp. 243–254, 2005.

41. S. Rezgui, G. M. Swift, R. Velazco, and F. F. Farmanesh, "Validation of an SEU simulation technique for a complex processor: PowerPC7400," in *IEEE Transactions on Nuclear Science*, vol. 49 I, no. 6, dec 2002, pp. 3156–3162.

42. S. Rezgui, R. Velazco, R. Ecoffet, S. Rodriguez, and J. R. Mingo, "Estimating Error Rates in Processor-Based Architectures," *IEEE Transactions on Nuclear Science*, vol. 48, no. 5, pp. 1680–1687, 2001.

43. P. K. Samudrala, J. Ramos, and S. Katkoori, "Selective triple modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs," *IEEE Transactions on Nuclear Science*, vol. 51, no. 5 IV, pp. 2957–2969, oct 2004.

44. A. Sánchez-Macián, P. Reviriego, and J. A. Maestro, "Hamming SEC-DAED and extended Hamming SEC-DED-TAED codes through selective shortening and bit placement," *IEEE Transactions on Device and Materials Reliability*, vol. 14, no. 1, pp. 574–576, 2014.

45. P. P. Shirvani, N. R. Saxena, and E. J. McCluskey, "Software-implemented EDAC protection against SEUs," *IEEE Transactions on Reliability*, vol. 49, no. 3, pp. 273–284, sep 2000.

46. ——, "Software implemented edac protection against seus," *IEEE Transactions on Reliability*, vol. 49, no. 3, pp. 273–284, Mar 2000.

47. F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam, "Mitigation of radiation effects in SRAM-based FPGAs for space applications," *ACM Computing Surveys*, vol. 47, no. 2, 2015.

48. J. Singh and J. Singh, "A comparative study of error detection and correction coding techniques," *Proceedings - 2012 2nd International Conference on Advanced Computing and Communication Technologies, ACCT 2012*, pp. 187–189, 2012.

49. L. Sterpone and M. Violante, "A new partial reconfiguration-based fault-injection system to evaluate SEU effects in SRAM-based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 54, no. 4, pp. 965–970, 2007.

50. A. Stoddard, A. Gruwell, P. Zabriskie, and M. J. Wirthlin, "A Hybrid Approach to FPGA Configuration Scrubbing," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 497–503, jan 2017.

51. K. Technologies, *Keysight Infiniium S-Series Oscilloscopes User's Guide*, 4th ed., 1400 Fountaingrove Parkway, Santa Rosa, CA 95403, Jan 2017.

52. T. Technologies, *DE1-SoC User Manual 1 www.terasic.com March 31, 2015*, 2015. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=816&FID=a9e8cb474881606fa975d2420a309fb6

53. J. Teifel, "Self-voting dual-modular-redundancy circuits for single-event-transient mitigation," *IEEE Transactions on Nuclear Science*, vol. 55, no. 6, pp. 3435–3439, 2008.

54. Terasic, *DE10-Standard User Manual*, 9F. No.176 Sec.2 Gongdao 5th Rd, Hsinchu City, Taiwan, Mar 2018.

55. J. L. Titus, "An updated perspective of single event gate rupture and single event burnout in power MOSFETs," *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 1912–1928, 2013.

56. U.S. Department of Defense, "Defense Space Strategy Summary," no. June, pp. 1–10, 2020. [Online]. Available: https://media.defense.gov/2020/Jun/17/2002317391/-1/-1/1/2020_DEFENSE_SPACE_STRATEGY_SUMMARY.PDF

57. M. Violante, L. Sterpone, M. Ceschia, D. Bortolato, P. Bernardi, M. S. Reorda, and A. Paccagnella, "Simulation-based analysis of SEU effects in SRAM-based FPGAs," in *IEEE Transactions on Nuclear Science*, vol. 51, no. 6 II, dec 2004, pp. 3354–3359.

58. J. Wang, "Radiation effects in FPGAs," *9th Workshop on Electronics for LHC*, vol. 900, p. 2, 2003. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.6269&rep=rep1&type=pdf%5Cnhttp://lhc-electronics-workshop.web.cern.ch/lhc-electronics-workshop/2003/plenaryt/wang.pdf

59. M. Wirthlin, "FPGAs operating in a radiation environment: lessons learned from FPGAs in space Related content Radiation effects in reconfigurable FPGAs Heather Quinn," *iopscience.iop.org*. [Online]. Available: https://iopscience.iop.org/article/10.1088/1748-0221/8/02/C02020/meta

60. Xilinx, "Radiation-Hardened, Space-GradeVirtex-5QV Family Data Sheet: Overview," vol. 192, no. v1.6, pp. 1–17, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds192_V5QV_Device_Overview.pdf

61. ——, "SP701 Evaluation Board," pp. 1–50, 2019.

62. Xilinx Inc., "UltraScale Architecture and Product Data Sheet : Overview Summary of Features RF Data Converter Subsystem Overview Soft Decision Forward Error Correction (SD-FEC) Overview," *Xilinx Technical Documentation*, vol. 890, pp. 1–46, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds889-zynq-usp-rfsoc-overview.pdf

63. M. Zheng, Z. Wang, and L. Li, "DAO: Dual module redundancy with AND/OR logic voter for FPGA hardening," *Proceedings of 2015 the 1st International Conference on Reliability Systems Engineering, ICRSE 2015*, pp. 1–5, 2015.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
12–01–2022	Master's Thesis	Sept 2020 — Mar 2022

4. TITLE AND SUBTITLE

Implementation and Characterization of AHR on a Xilinx FPGA

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Dittrich, Andrew J, Capt, USSF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering an Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-22-M-025

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Intentionally Left Blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Adaptive-Hybrid Redundancy (AHR) architecture was modified and tested in hardware using Commercial-Off-The-Shelf (COTS) Field-Programmable Gate Arrays (FPGAs). The AHR architecture mitigates the effects that the Single Event Upset (SEU) and Single Event Transient (SET) radiation effects have on processors and was tested on a Microprocessor without Interlocked Pipeline Stages (MIPS) architecture. The AHR MIPS architecture was implemented on two Xilinx FPGAs using a serial based communication network. The runtime performance of AHR MIPS was measured and compared against the performance of TMR and TSR MIPS. AHR MIPS demonstrated flexible runtime performance that was nearly as fast as TMR MIPS, never as slow as TSR MIPS, and demonstrated performance in between those extremes. Hardware testing and verification of AHR MIPS showed that the AHR mitigation strategy presents a large performance tradespace, where a user can adjust both the runtime processor performance and radiation tolerance to fit the constraints of a space mission.

15. SUBJECT TERMS

Hybrid Redundancy, Adaptive Redundancy, SEU, SET, FPGA, MIPS, Radiation, Radiation-tolerant, Xilinx

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maj. Nicolas Hamilton, AFIT/ENG
U	U	U	UU	227	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x4220; nicolas.hamilton@afit.edu