ADAPTIVE-HYBRID REDUNDANCY FOR RADIATION HARDENING

DISSERTATION

Nicolas S. Hamilton, Major, USAF

AFIT-ENG-DS-19-S-005

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-DS-19-S-005

ADAPTIVE-HYBRID REDUNDANCY FOR RADIATION HARDENING

DISSERTATION

Presented to the Faculty

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 Doctor of Philosophy in Electrical Engineering

Nicolas S. Hamilton, M.S.

Major, USAF

September 12, 2019

AFIT-ENG-DS-19-S-005

ADAPTIVE-HYBRID REDUNDANCY FOR RADIATION HARDENING

DISSERTATION

Nicolas S. Hamilton, M.S.
Major, USAF

Committee Membership:

Scott R. Graham, Ph.D.
Chair

Major Timothy J. Carbino, Ph.D.
Member

James C. Petrosky, Ph.D.
Member

Major J. Addison Betances, Ph.D.
Member

AFIT-ENG-DS-19-S-005

Abstract

An Adaptive-Hybrid Redundancy (AHR) mitigation strategy is proposed to mitigate the effects of Single Event Upset (SEU) and Single Event Transient (SET) radiation effects. AHR is adaptive because it switches between Triple Modular Redundancy (TMR) and Temporal Software Redundancy (TSR). AHR is hybrid because it uses hardware and software redundancy. AHR is demonstrated to run faster than TSR and use less energy than TMR. Furthermore, AHR allows space vehicle designers, mission planners, and operators the flexibility to determine how much time is spent in TMR and TSR. TMR mode provides faster processing at the expense of greater energy usage. TSR mode uses less energy at the expense of processing speed. AHR allows the user to determine the optimal balance between these modes based on their mission needs and changes can be made even after the space vehicle is operational. Radiation testing was performed to determine the SEU injection rate for simulations and analyses. A Field Programmable Gate Array (FPGA) was used to expedite testing in hardware.

For my wife and kids

Acknowledgements

First and foremost, I am thankful for the opportunities in life that God has blessed me with so that I have been able to arrive at the point in my life where I have been afforded the opportunity to earn a PhD. I am also thankful to my wife and children who have supported me in my work, particularly for my wife kicking me out of the house and making me work from work rather than from the home office where I was not able to get much done. I would also like to thank Dr. Graham and Maj Carbino who both served as my primary advisors at different times and have guided me in my research. I also thank Dr. Petrosky and Maj Betances for the invaluable insights they provided into radiation effects and the intricacies of VHDL design respectively. Finally, the great team at Sandia National Laboratories Ion Beam Lab were instrumental in helping me perform radiation testing, without which I could not make the necessary radiation comparisons to previous works.

Nicolas S. Hamilton

Table of Contents

Page

Page

List of Figures

List of Tables

ADAPTIVE-HYBRID REDUNDANCY FOR RADIATION HARDENING

I. Introduction

1.1 Research Context

Space electronic components experience radiation effects at a much higher rate than terrestrial components, which are protected by the Earth's atmosphere. These effects disrupt low level computing elements in unpredictable ways and must be mitigated if the computed results are to be trusted. The typical approach is to apply some form of radiation hardening to the processors. However, this is a significant effort amortized over fewer processors, making these processors expensive. In addition, it takes years to develop a radiation hardened version of a processor, therefore a radiation hardened version of a state-of-the-art processor is necessarily two or more generations behind commercially available state-of-the-art processors.

Space data processing has traditionally been performed on the ground because of computing limitations; however, newer sensors generate significantly more data than current bandwidth limitations allow. As a result, space vehicle designers are interested in solutions to perform more processing on-board space vehicles.

Another consideration when using radiation hardened processors is that they typically use more power than their commercial counterparts, and when incorporated into a space vehicle, require the space vehicle to provide more power generation and energy storage in the form of larger solar panels and batteries. Larger solar panels and batteries directly increase the cost of the space vehicle. The added weight of the larger solar panels and batteries directly increase the vehicle's launch costs.

Researchers and space vehicle designers seeking to increase space based processing capability and reduce power requirements, vehicle weight, and launch costs are turning to commercial-off-the-shelf (COTS) state-of-the-art processors and Field Programmable Gate Arrays (FPGAs). Rather than hardening these technologies directly, designers may be able to mitigate errors caused by radiation through on board redundancy. One way to accomplish this is by running multiple processors in parallel and using a voter to determine the correct output. Alternatively, redundancy could be incorporated in the software. These approaches are discussed in Chapter II.

The motivation for using these approaches is to field space systems sooner and at lower cost. This is especially true for United States Air Force satellites tasked with command, control, communications, computers, intelligence, surveillance, and reconnaissance (C4ISR) missions. The Air Force often designates these types of systems as being in the interest of national security and as such, follow the most stringent radiation hardening protocols. According to a 2014 article in Space News, GPS III satellites cost about $547 million each [32]. If Air Force satellites could utilize COTS processors and FPGAs, the Air Force could potentially save millions of dollars on development and procurement of radiation hardened processors for future C4ISR and GPS satellites.

FPGAs and COTS processors share some advantages and disadvantages, but differ in others. One significant difference is that FPGAs can be reconfigured at any time, including after deployment. Both are susceptible to Single Event Upsets (SEUs) and Single Event Transients (SETs), though in slightly different ways. FPGAs are more vulnerable to SEUs which affect configuration memory. They are also vulnerable to SEUs and SETs affecting user logic. COTS processors are vulnerable to SEUs and SETs affecting their registers. FPGAs and COTS processors both suffer from Total Ionizing Dose (TID) and Enhanced Low Dose Rate Sensitivity (ELDRS) effects which

cause permanent damage over time. This research does not examine the hardening techniques designed to mitigate TID and ELDRS, but instead focuses on SEU and SET mitigation. Furthermore, this research does not examine hardening through design changes at the transistor level or shielding of FPGAs and COTS processors but rather on redundancy.

FPGA SEU and SET vulnerabilities are minimized when implementing mitigation strategies such as internal scrubbing of configuration memory and user logic redundancy. COTS processors also utilize logic redundancy, but not internal scrubbing because they do not have configuration memory. This research focuses on mitigation strategies employing redundancy in much the same way as previous approaches discussed in Chapter II. The main difference between this research and previous research is that previous research typically implemented a single redundancy method which could not be changed once implemented on a space vehicle. This meant that the space vehicle was constrained to the power and performance penalties incurred by the redundancy method for the duration of its mission.

Two such redundancy methods are triple modular redundancy (TMR) and temporal software redundancy (TSR). Each redundancy method has its benefits and drawbacks. This research seeks to enhance the benefits and minimize the drawbacks by switching between redundancy techniques in real-time depending upon the radiation environment, processor loading, and energy consumption. This can be achieved by using a controller to switch between redundancy methods. This flexible form of redundancy is called Adaptive-Hybrid Redundancy (AHR) and provides satellite designers tradespace between energy consumption, and time to complete a processing task. The ability to switch between redundancy methods makes this approach adaptive while the use of both hardware and software redundancy methods makes it hybrid. The AHR processor proposed in this research switches between TMR and

TSR.

1.2 Assumptions

To maintain focus and scope, the following assumptions are made throughout the remainder of this dissertation.

1. The only radiation effects considered are Single Event Upsets (SEUs) and Single Event Transients (SETs)
   (a) All other radiation effects do not occur
   (b) Multiple-bit upsets (MBUs) where a single radiation strike causes errors in two or more adjacent registers are highly unlikely and do not occur
2. The processor refers only to the Controller and Datapath of a processor
   (a) The Controller consists of a finite state machine with a state register, instruction register, and instruction decoding logic that translates instructions into control signals for the Datapath
   (b) The Datapath consists of general purpose registers (GPRs) that store data to be processed, a program counter (PC) register, logic to update the GPRs and PC register, an arithmetic logic unit (ALU) used to process data, and logic to control the disposition of processed data. The function of the Datapath is controlled by control signals from the Controller
   (c) Modern processors are typically super-scalar and make use of combinational logic and many additional registers for branch prediction and other applications. The additional combinational logic and registers used by super-scalar processors are also considered to be part of the processor

3. The processor and all components therein are subject to SEUs and SETs with few exceptions

   (a) The TMR Voter in TMR MIPS described in Section 3.3 is immune to errors

   (b) The AHR Controller in AHR MIPS described in Section 3.5.1 and the multiplexers used by the AHR Controller for signal routing described in Section 3.5.2 are immune to errors

4. Memory refers to the location where instructions and data are stored prior to being read by the processor. It is also the location where the processor may write data. In practical applications, this memory may be cached memory, random access memory (RAM), read only memory (ROM), a hard disk drive (HDD), or a solid-state drive (SSD)

5. Memory is immune to all errors

   (a) Memory hardening is not implemented in this research

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

1.3 Research Questions

This research seeks to answer the following questions:

1. Can multiple redundancy methods be incorporated into the redundancy design?

2. Is it possible to allow flexibility in redundancy methods for the duration of a space vehicle's lifetime?

3. Is it possible to switch between these methods based on mission needs?

4. What are the timing and energy tradespaces available to a designer, mission planner, or operator?

1.4 Dissertation Organization

The remainder of this document is organized as follows. Chapter II provides background for this research. Chapter III develops the AHR architecture. Chapter IV evaluates AHR against TMR, TSR, and an unmitigated processor in a perfect environment with no errors present. Chapter V discusses how an error rate and error injection method were devised and mathematical tools were developed to evaluate the performance of AHR when compared to TMR and TSR when subjected to errors. Chapter VI presents the results of evaluating AHR against TMR and TSR when errors are injected. Chapter VII presents conclusions and contributions of this research as well as suggestions for future work.

II. Background

2.1 Introduction

This chapter lays the foundation for the development of Adaptive-Hybrid Redundancy (AHR). It begins with a discussion of radiation effects in Section 2.2 that delves into radiation effects known as Single Event Effects (SEEs) with an emphasis on Single Event Upsets (SEUs) and Single Event Transients (SETs). The discussion continues with the mitigation of those radiation effects in Section 2.3, which provides an overview of many mitigation strategies that have previously been applied in radiation environments. Section 2.4 examines a method for making comparisons between orbital radiation environments and experimental radiation environments with the goal of making a comparison between a particular flight experiment and an experiment performed in this research.

These first three sections are followed by a brief introduction to the hardware upon which AHR is implemented for this research in Section 2.5. Finally, Section 2.6 discusses previous methods to evaluate radiation vulnerability and redundancy techniques and leads to the selection of the evaluation methods used in this research.

2.2 Single Event Effects

SEE is a broad term applied to effects caused by radiation striking a computer processor which have immediate impacts on the internal state and outputs. (In contrast, there are also effects caused by long term radiation exposure which do not immediately affect the internal state or output of the processor, which are also problematic, but outside the scope of this research.) While all SEEs have an immediate impact or effect, some impacts may be permanent, such as Single Event Gate Rupture (SEGR), Single Event Burnout (SEB), or Single Event Latchup (SEL). SEL can be

corrected before device failure if a gate's power is cycled, but SEGR and SEB cannot be corrected this way. Still others are truly transient, but may have permanent computational or program effects if left unmitigated. These include the SET, SEU, and Single Event Functional Interrupt (SEFI) [17, 26, 65, 108]. Because they are more common, and because this research is primarily concerned with onboard mitigation techniques, this chapter will focus on SEUs and SETs; however, the related categories of SEGR, SEB, SEL, and SEFI are included to complete the discussion on SEEs.

2.2.1 Permanent Single Event Effects

Both the SEGR and the SEB are caused when radiation strikes a transistor with sufficient energy to cause immediate and permanent failure. SEGR results from a failure of the insulating layer between the gate and depletion region of the transistor; which effectively destroys the gate. In contrast, SEB arises when radiation activates some parasitic component inherent in the chip. In each of these categories of radiation events, the effects are permanent and non-recoverable [26, 65].

2.2.2 Semi-Permanent Single Event Effects

The mechanism causing Single Event Latchups (SELs) is distinct from the mechanism that causes SEGR, SEU, and SET, but somewhat similar to SEB. To better understand SELs, consider that some computer chip technologies make use of CMOS gates where the silicon comprising the gates are placed on top of a doped base material. This doped base material allows for the formation of parasitic P-N-P-N transistors between NMOS and PMOS transistors. In normal operation, these parasitic transistors are off, but a radiation strike has the potential to activate them. Once turned on, these parasitic transistors form a positive feedback loop that causes the parasitic transistors to draw more current. If the feedback loop is not broken, the

logic gate could burn out and fail. Some CMOS technologies have current sensors that detect current spikes caused by SELs; then turn off power to the logic gate or a portion of the computer chip where the logic gate is located. Turning off the power stops the positive feedback loop and prevents the SEL from becoming destructive. Power is then restored and normal operation resumes. SELs can also be prevented by using silicon on insulator CMOS technology or other means of isolating NMOS and PMOS transistors so that parasitic transistors cannot form [26, 65, 108].

Single Event Functional Interrupts (SEFIs) can be the result of SETs that are stored in registers and SEUs that propagate throughout a computer processor over several clock cycles. The propagated errors influence various intermediate and final results that the processor is tasked with computing. They may also affect the flow of a program that is running on a processor if that program contains any branch or jump instructions that are impacted by SEU and SET propagation. SEFIs may also occur when radiation triggers a built-in-self-test mode or a reset of a processor [17, 26, 65, 108].

2.2.3 Transient Single Event Effects

Single Event Transients (SETs) are caused when ionizing radiation produces a voltage and current pulse in a transistor. That pulse has the capability to propagate energy through neighboring combinational logic. The pulse may travel through several, or only a few gates, depending upon a number of factors. First, the pulse may be attenuated or amplified depending upon the parameters of the gates through which the pulse passes (i.e., length, width, depth, doping concentrations, type of gate, etc.). Second, the pulse may be logically masked, which can occur when a logic 1 pulse encounters an AND gate where another signal is a logic 0. Because both inputs would have to be a logical 1 for the output to change, the erroneous logic 1

pulse has no further effect. Third, the pulse may be temporally masked; this can occur when a pulse reaches the input to a memory element, such as a register or flip-flop, but does not arrive with sufficient voltage and duration during the memory element's setup and hold time. On the other hand, if the pulse is not masked, it may be stored in one or more registers, and will likely result in incorrect computation [4, 5, 15, 24, 26, 41, 60, 65, 81, 89, 90, 108].

Single Event Upsets (SEUs) are caused when ionizing radiation strikes a transistor that is part of a memory element with sufficient charge to directly flip that memory element. In other words, the radiation causes a memory element storing a logic 0 to store a logic 1 or vice versa [8, 16, 17, 26, 33, 65, 105, 108]. While this is a problem generally, SEUs can be particularly bothersome for Field Programmable Gate Arrays (FPGAs). FPGAs often utilize SRAM cells in their configuration memory to instantiate a user design. They govern the routing of signals as well as the implementation of specific logic functions. Beyond merely changing the value of an operand, configuration memory SEUs can alter the users desired functional logic and even create short circuits. At a minimum, these errors have the potential to prevent the FPGA from accomplishing its designed task. In extreme cases, configuration logic SEUs may permanently damage the FPGA.

2.3 Mitigating SEUs and SETs

There are numerous methods through which SEUs and SETs may be mitigated. These include shielding, hardware modifications, software modifications, and redundancy.

Hardening can be achieved through physical shielding at the circuit level, system level, and various levels in between. The ability to shield at various levels depends on how much is known about the layout of a processor. Shielding could be applied to

specific critical circuits like power transistors that ensure proper power distribution or transistors tasked with clock distribution; however, specific circuit shielding only works if the exact physical locations of circuits are known. Shielding can also be applied to an entire processor by placing metal over the processor. Physical shielding is not within the scope of this research as this research focuses on hardening through redundancy.

Hardware modifications, software modifications, and redundancy for SEU and SET mitigation are further discussed. These modifications can be divided into two distinct levels. The first is circuit level hardening and the second is system level hardening. At the circuit level, hardening is achieved by making changes to transistors or adding components to combinational logic and memory elements. System level hardening is achieved through redundancy of components at the logic gate level, system level, or any level in between. System level hardening may also be achieved through software redundancy. Circuit level hardening is discussed first, then system level hardening.

2.3.1 Circuit Level Hardening

Some methods of circuit level radiation hardening make changes to the material properties of the transistor. Hughes et al. discuss a radiation hardened gate made of heavily doped n-type silicon where they built and tested gates before and after irradiation and demonstrated positive results [43]. Uemura et al. make changes to the transistors' physical dimensions and also create cancellation regions between transistors to "cancel out" trapped charges that would otherwise cause a multi-node upset. They also discuss the use of triple well versus double well technologies in their mitigation approach [102].

Other methods of circuit level radiation hardening require changes to the circuitry

comprising combinational logic and the memory cells. One such modification is the introduction of a special filter into combinational logic to mitigate SETs. These filters prevent short pulses from passing, but allow longer pulses to pass [82]. This effectively eliminates radiation induced pulses while permitting desired signals to pass. Filters can also be added in memory elements to prevent bit flips caused by SEUs. Filters can be composed of delays created by inverter chains coupled with a guard gate or a simpler filter comprised of resistors and/or capacitors [3, 17, 29, 64, 82, 85]. Another modification changes the structure of memory elements to mitigate SEUs. One example modification of the simple SRAM cell is the Dual Interlocked storage Cell (DICE) [11], which effectively blocks transient pulses from causing upsets to the memory cell. Another SRAM cell modification, called the Soft Error Interception Latch [47], implements multiple delay elements and gates to stop any pulses from propagating and causing an upset to the memory cell. Unfortunately, adding filters and changing the basic structure of the SRAM cell unavoidably require costly and time consuming processor redesign as well as changes to the fabrication process. The addition of filters also typically cause an increase in the amount of time required to store a value to the SRAM cell.

2.3.2 System Level Hardening

System level hardening through redundancy can be achieved in hardware, software, or a hybrid of hardware and software. Hardware redundancy can take place at the gate level, the processor level, or somewhere in between. Hardware redundancy may use multiple copies of specific circuits or subcircuits where inputs are driven by a single source and outputs can be compared to determine the correct result for that circuit. Hardware redundancy may also employ error correcting codes in various stages. Some hardware redundancy strategies make use of both, and they are fur-

ther discussed in the subsections on Hardware Redundancy and Hybrid Redundancy. Software redundancy typically performs a single instruction on a processor multiple times and compares the results of each iteration to determine the correct result to forward to the next stage. Hybrid designs make use of both hardware and software redundancy; some designs will place more emphasis on the hardware or software while others will utilize both equally. The following pages provide specific examples of various hardware, software, and hybrid redundancy strategies.

2.3.2.1 Hardware Redundancy

Multiple Copies of Circuits or Subcircuits This section discusses some examples of hardware redundancy where there are multiple copies of circuits or subcircuits.

The first example is of redundant registers to store the processor's state. The processor's state consists of all user defined registers, the program counter, and the instruction register. The registers are assumed to be radiation hardened and immune to SEUs while the processor's combinational logic is vulnerable to SETs. At any given time, the current state is being written to one set of the redundant registers while the other set of registers stores the previous state. If a SET occurs, it may impact the set of registers to which the processor is currently writing data. The SET can be detected by the processor, which is able to continue processing using the set of registers that hold the previous state. The set of registers which were potentially impacted by the SET are then overwritten when the processor processes the data from the previous state and stores the updated state [34].

The second example illustrates the use of dual modular redundancy (DMR). DMR makes use of two processors and a single comparator/controller. Both processors run in lockstep (meaning that they simultaneously receive identical inputs, including the

clock) and produce outputs that are checked by the comparator. The controller periodically interrupts the two processors to create a save/restore point in memory that saves the processor's internal state at a particular moment in time. If the two processors ever disagree on their outputs, the comparator/controller can then go back by restoring the internal state of both processors to the most recent save/restore point. (This approach assumes that the save/restore point is saved in radiation hardened memory [20, 22, 27].) In some implementations, the two processors are implemented differently (i.e. two different chips from the same manufacturer or different manufacturers), but designed to produce identical results [95]. Figure 1 provides a simple illustration of DMR.

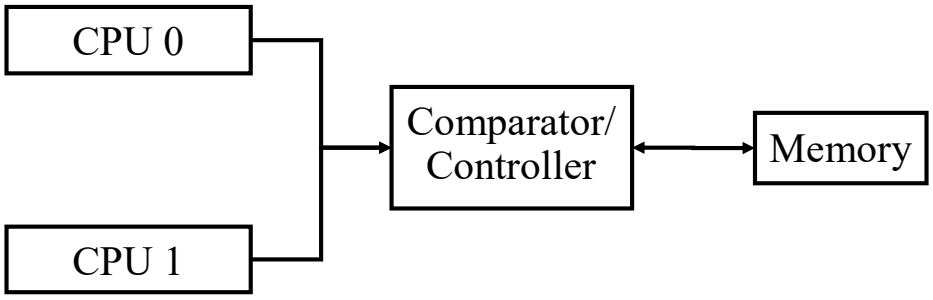

Figure 1. Dual Modular Redundancy Simplified Block Diagram

A different method of DMR makes use of two processors that do not run in lockstep. In this example, two cores of a four core processor are utilized such that one core runs a program and the second core runs the same program, but a fixed number of instructions behind the first core. The first core provides values loaded from memory to the second core so the second core does not have to load them from memory. The first core also provides the second core the outcome of its branch decisions. The second core compares the address information provided by the first core to its own as well as the first core's branch outcomes to its own branch decisions to

determine if an error has occurred. In the event that the program needs to store data to memory, the store only occurs after the second core processes the store instruction and determines that the address and data to be written to memory match those of the first core [30].

Another approach uses two processors, similar to DMR, but only one processor runs the program while the second processor acts as a watchdog to detect errors. The watchdog monitors the main processor's inputs and outputs to detect errors in the flow of the program and the data being written to memory [58]. While the watchdog approach cannot detect every error that DMR detects, it is able to detect errors at a lower cost in terms of size of the second processor and power used by the second processor. In one particular watchdog approach, signatures are embedded into the program so the watchdog can determine whether the program flows correctly (i.e. ensures no illegal/incorrect program branches are taken). The main processor is designed to ignore the embedded signatures while the watchdog processor compares the embedded signatures to the ones it computes in real time [72].

The next example uses Triple Modular Redundancy (TMR) at a low level. An ARM Cortex-R4 processor was modified such that all flip-flops were replaced with three copies of the same flip-flop, followed by a voter. Different delay elements were added at the inputs to the flip-flops to reduce the probability of a SET causing more than one of the three flip-flops to store an erroneous value [74].

In another TMR example, TMR is implemented at the system level. In this configuration, three copies of the same processor operate on the same inputs and their outputs are examined by a voting circuit. All three processors produce the same outputs when there are no errors. If the output of one of the three processors differs, it is assumed to have encountered an error, and the correct output is determined by majority vote. The voter circuity resets the "incorrect" processor and sets the

processor's internal state to match the internal state of the two correct processors. If all three processors disagree on the output (i.e. no majority exists), the voter resets all three processors and restores them to a previously saved internal state (the TMR restoration process is identical to the DMR restoration process). This approach also assumes that the save/restore point is saved in radiation hardened memory [6, 7, 10, 45]. TMR is such a commonly used mitigation technique that some FPGA manufacturers incorporate TMR into their hardware documentation as a means to harden their processors [12]. TMR is considered the "gold standard" by the U.S. Air Force for protecting electronics aboard spacecraft and launch vehicles. Figure 2 provides a simple illustration of TMR. This research uses this TMR approach in the development of Adaptive-Hybrid Redundancy (AHR). This is the simplest of the TMR approaches presented in this chapter, which makes it an ideal candidate for evaluating AHR. TMR is selected over DMR because TMR is considered the "gold standard" for U.S. Air Force space systems.

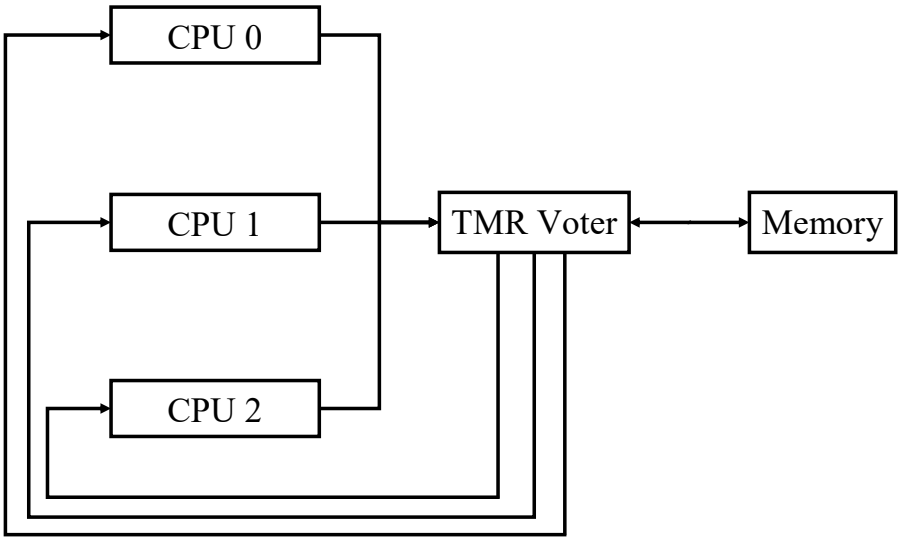

Figure 2. TMR MIPS Simplified Block Diagram

While full TMR is very effective, it is also costly in terms of energy. The next TMR approaches described are more energy conscious. In one approach, TMR is only implemented when radiation levels exceed some threshold, as measured by an external radiation sensor. However, when radiation levels are below the threshold, a single processor is used, thereby minimizing power consumption [55]. In other approaches, TMR is selectively implemented for critical subcircuits on an FPGA or critical processors in a larger computer system rather than implementing TMR for every subcircuit or an entire processor. [31, 91]. These approaches reduce power consumption as fewer components are triplicated.

Another TMR approach has no independent voting circuitry. In this example, all three processors compare the outputs of the other two processors to their own outputs to detect errors. Upon detecting an error, the processors execute exception handling code. One of the processors is designated as the "master" processor, and when an

error occurs, the first processor to complete error handling becomes the master and reboots the other two. If the new master fails to correctly reboot the other two processors, the next processor to finish exception handling detects this and becomes the master and reboots the other two processors. The "master" processor loads its own internal state into the two rebooted processors to ensure all three processors have the same internal state before resuming normal operations [56].

Hardware redundancy may also be achieved by using TMR and NMR (N-Modular Redundancy) systems where permanently failed processors are replaced with spares. NMR is like TMR where there are N processors rather than three processors and outputs are decided by majority vote. Replacements are needed when a processor fails due to Total ionizing Dose (TID), Enhanced Low Dose Rate Sensitivity (ELDRS), SEB, SEGR, SEL, or some other permanent damage. In this way, a TMR system can continue with three functioning processors or an NMR system with N functioning processors even after a permanent failure. As in the previous TMR with voting example, temporary faults are corrected by copying the system state of the correct processors to faulty processors [87].

Another NMR system uses a total of four processors where two processors are on one FPGA and two processors are on another FPGA. All four processors receive the same inputs. The outputs of the two processors on the same FPGA are compared. Next, the output of the two FPGAs are compared. At this stage, processors in error are identified. Because the number of processors is even, if one pair of processors reach one result and another pair of processors reaches a different result, error recovery must follow the same approach as if all processors are in error as discussed for TMR. If two processors agree on a result and the other two processors have differing results, the two that agree are assumed to be correct and the internal states of the other two processors are corrected to match the internal states of the correct processors [21, 94].

Yet another distinct approach is halfway between DMR and TMR. In this approach, two "main" processors proceed in lockstep, periodically saving checkpoints, until an error occurs at a checkpoint. The two main processors proceed to another checkpoint after the one at which the error occurred while a third processor starts from a prior, error-free checkpoint. The third processor proceeds past the checkpoint at which the error occurred and catches up with the main processors at the following checkpoint. All three processors are compared and the erroneous processor can be identified and corrected if the third processor matches one of the main processors. The third processor is turned off after this checkpoint. If there is no agreement between the third processor and the main processors at this checkpoint, the main processors are restored to the last error-free checkpoint [75].

Error Correcting Codes Error correcting codes (ECCs) began with simple parity bits; these bits indicated whether there were an even or odd number of logic 1s in a byte or other grouping of bits (i.e. 16-bit half-word, 32-bit word, and etc.). Unfortunately, parity bits can only detect when an odd number of bits are in error. If there are an even number of errors, the errors go undetected. Additionally, the number of errors and the location of the error or errors is unknown [10]. Newer ECCs incorporate multiple bits such that the number of errors and location of the errors is precisely known.

The most common ECCs can detect and locate two errors while being able to correct a single error through the use of combinational logic. These codes are commonly referred to as single error correcting-dual error detecting (SEC-DED) codes [10, 54, 93] Some of these codes can be expanded to detect and correct more errors.

Other ECCs are designed to survive addition and multiplication operations. These codes include AN (cyclic code with ring of integers modulo $2^n - 1$ for some n), Bose-Chaudhuri-Hocquengham (BCH) codes, Modified Reflected Binary (MRB) code, num-

ber theoretic transforms, and residue codes [10, 19, 40, 48, 52, 53, 54, 76].

While ECCs are more commonly used to protect memory, they may also be used to protect registers in a processor. In one example, ECCs were used to only protect the most vulnerable registers while parity bits were used to detect errors in the remaining registers [50].

2.3.2.2 Software Redundancy

Redundancy achieved through software is also referred to as Temporal Software Redundancy (TSR) and is characterized by duplicating instructions, spreading them out over time, and comparing results at critical points in a program.

One simple implementation of software redundancy is called Error Detection by Duplicated Instructions (EDDI) and duplicates all instructions except for store instructions [71]. The instructions are duplicated such that the original instruction and its duplicate use different registers for the arguments and different registers for the results. The use of different registers means that the outcomes of the duplicated instructions are independent of one another in terms of radiation events. This implementation also adds a check prior to every store instruction. This check ensures a match between the value to be stored to memory and the values' duplicate along with the address to which it is to be stored and the address' duplicate. If the values or addresses do not match, an error has occurred and error recovery must begin. If both values and addresses match, the store word instruction is executed [71]. This is better explained by looking at the following example in Table 1 which compares a simple program with a program incorporating EDDI.

Table 1. Simple Software Redundancy Example

Instruction Number	Original Set	Redundant Set
1	LUI R1 1	LUI R1 1
2	LUI R2 2	LUI R15 1
3	ADD R3 R1 R2	LUI R2 2
4	SW R3 R0 OFFSET	LUI R16 2
5		ADD R3 R1 R2
6		ADD R17 R15 R16
7		BNE R3 R17 ERR
8		SW R3 R0 OFFSET

In this example, LUI R1 1 is a load upper immediate instruction that loads the immediate value, 1, into the upper 16-bits of a 32-bit register named R1. ADD R3 R1 R2 is an addition command that adds the contents of registers R1 and R2 and stores the sum in register R3. SW R3 R0 OFFSET is a store instruction that stores the contents of register R3 to the memory location specified by R0 plus a constant offset specified by OFFSET. The register R0 always contains the value 0. The original program loads 1 and 2 into registers R1 and R2 so that they contain the values $1 \cdot 2^{16}$ and $2 \cdot 2^{16}$ respectively, adds the two values and stores the result of $3 \cdot 2^{16}$ to R3, and stores R3 to memory location OFFSET. The redundant program does the same thing as the original set until the add instruction, but does it twice. The second time it performs each operation, the results are stored in different registers. If no errors occur, R15 = R1, R16 = R2, and R17 = R3. Additionally, the values of R15, R16, and R17 do not depend upon the values of R1, R2, and R3 because they were computed independently. The next big difference is the comparison performed on line 7. The BNE R3 R17 ERR instruction is a branch if not equal instruction. The instruction decides that if R3 does not equal R17, code execution should jump to some error recovery instructions at the location specified by the branch distance as ERR. If R3

does equal R17, instruction 8 is performed which stores R3 to the memory location specified by OFFSET.

Each register and its duplicate must be maintained so long as it is needed by future instructions. For example, if R1 and R15 are used to compute another value to be stored in R4 and R18, R1 and R15 must not be overwritten. One particular question asked why the redundant set could not perform original instructions 1, 2, and 3 followed by original instructions 1, 2, and a modified form of redundant instruction 6 "ADD R17 R1 R2". This would still achieve the desired independence between R3 and R17 for the comparison of redundant instruction 7; however, R1 and R2 would have no duplicates and be unprotected from errors should R1 and R2 be needed for future instructions. Additionally, the LUI instruction is the only Basic MIPS instruction that does not require any previously computed values stored in another register. If R1 and R2 were assigned as the result of an ADD, AND, SUB, or any other instruction, recomputing R1 and R2 based on the same values supplied by the register arguments to the ADD, AND, or SUB instruction would not make them independent, and therefore susceptible to identical errors.

This research uses the EDDI TSR approach in the development of Adaptive-Hybrid Redundancy (AHR). This is the simplest of the TSR approaches that will be presented in this chapter, which makes it an ideal candidate for evaluating AHR. While more complicated forms of TSR could be implemented, it could obfuscate some of the results such that the advantages of AHR over TMR or TSR alone would not be readily apparent.

Some implementations of this approach created a compiler that would automatically convert a non-duplicated program like the one shown in Table 1 to a redundant set like the one shown in Table 1 [71, 101]. Oh's specific implementation also sought to minimize the time penalty incurred by the additional instructions by making use

of a pipelined architecture in a super-scalar processor [71]. While Oh et al. and Tokponnon et al. did not specify the nature of the error recovery instructions; it is worth noting that duplication more than doubles the total number of instructions when the additional error recovery instructions are added. Oh and McCluskey also developed a method that does not duplicate all instructions in order to realize energy savings [69].

Oh et al. further improved upon their work by adding signature detection instructions to ensure that programs were followed sequentially and that branch instructions were executed properly [70]. These signature detection instructions ensured that errors affecting the program counter (PC) were detected. When combined with instruction duplication, a more robust software redundancy method called SWIFT was created [79]. This was further strengthened by SWIFT-R which modifies SWIFT by triplicating all instructions rather than duplicating all instructions. This is essentially software implementation of TMR with additional protections for PC errors [80].

Reis et al. also discussed two additional software redundancy methods in the same paper. In the one referred to as trump, an instruction duplication method creates duplicate instructions that are AN-encoded where AN-encoding could be applied [80]. The other, referred to as Mask, "enforces statically known invariants to eliminate faults that can be reasoned away. Using these invariants, Mask can remove faults that would otherwise be deleterious, thus increasing reliability without redundant execution." [80]. The paper goes on to discuss hybrids between SWIFT-R, trump, and Mask. SWIFT-R and SWIFT-R/trump provide the greatest protection but have normalized run times that are nearly twice as long as a program that implements no redundancy. The trump and trump/Mask methods provide minimal protection and have normalized runtimes that are 36% longer than an unprotected program.

Software signature detection has also been used purely to detect illegal branches

while not simultaneously protecting against data errors. In one particular example, a program is broken into blocks and each block is assigned a parity value that depends upon its predecessor block. This parity value is computed at compile time. Signature checking instructions are inserted into the blocks that also compute the parity values at runtime. If runtime and compile time signatures do not match, then an illegal branch has occurred in the code. This particular example provides no recovery method when an error is detected [103].

2.3.2.3 Hybrid Redundancy

Hybrid redundancy refers to any amalgamation of one or more redundancy methods, and more particularly if those methods are fundamentally different. Hybrids can include hardware/software, hardware/ECC, software/ECC, hardware/software/ECC, methods that combine processor and memory hardening, and many more.

The first example of hybrid redundancy is specific to FPGAs and cannot be applied to other computer processors because the mitigation approach focuses upon SEUs affecting an FPGA's configuration memory. The primary means by which FPGA designers can overcome configuration memory SEUs is by implementing internal scrubbing. Internal scrubbing is the process where configuration memory is reprogrammed from an external, radiation hardened source to correct any errors that occur. This process takes time and must be performed periodically to ensure errors are corrected as soon as possible after they occur. However, errors can still occur between internal scrubbing intervals. To ensure these errors do not cause problems, additional mitigation is needed in the form of user logic redundancy. By making user logic redundant, a configuration error affecting one of two or more redundant user logic components will not be able to alter the desired system state or outputs. Redundancy gives the periodic internal scrubbing an opportunity to correct configuration

errors before there are enough errors to affect desired system state or outputs. One approach utilizes internal scrubbing with dual redundancy of user logic [23]. This mitigation approach also uses dynamic partial reconfiguration (DPR) instead of periodic internal scrubbing of the entire configuration memory [23]. DPR consists of constantly reading the configuration memory to detect errors, then only reconfiguring portions of configuration memory that contain errors [92]. DPR saves a significant amount of time and energy [23]. TMR for user logic in concert with internal scrubbing is another common strategy [57, 65, 73]. Another study paired TMR with DPR to take advantage of the ability of TMRs majority voting scheme to process faster than DMR (which has to return to a previously saved system state every time it encounters an error) and the time and energy savings provided by DPR.[59].

The next example of hybrid redundancy uses TMR (or NMR) for computer processors and ECCs to protect memory. A majority voting system is in place so the processors in agreement (2 of 3, 3 of 5, and etc.) determine the correct outputs and processors that are in error are periodically reset so that their states match the state of the processors in agreement [42]. A very similar approach selectively implements TMR for critical circuits to minimize the overall impact of SETs while still utilizing an ECC to protect memory [88].

Another approach performs a single calculation twice on the processor and uses a radiation hardened comparator to compare the results. If there is no match, the calculation is re-performed twice more. The first calculation of each round is compared and the second calculation of each round is compared. The ones that match are assumed to be the correct result [13]. This is a hybrid between software instruction redundancy and hardware radiation hardening.

Czajkowski also discusses redundancy achieved through temporal and spatial redundancy by placing duplicated instructions on different cores of a single processor

and separated by a clock cycle. If the results of the two instructions do not match, the instruction is carried out a third time on a third core. If the results of two out of three executions of the same instruction match, processing continues to the next instruction, if not, recovery operations are started [14].

One hybrid method uses a primary processor to run programs and a second processor that checks the first for faults. The second processor only checks certain error prone portions of the primary processor. Software recovery is used to restore the primary or secondary processor when a fault is detected. This method is an alternative to the DMR approach, but is smaller and more energy efficient [83].

Another hybrid technique implements NMR in virtual processors running in lockstep in the same way that NMR runs on physical processors. The approach examined the differences between DMR, TMR, and quadruple modular redundancy (4 processors). The voter and recovery operations are also implemented as virtual processes [46].

Reinhardt and Mukherjee discuss three different architectures. The first detects errors by comparing outputs of two identical threads running at the same time on different processors [78]. The second compares the outputs of two identical threads running at different times on the same processor [78, 104]. Vijaykumar et al. improved this second architecture by adding error recovery so that errors are corrected rather than only being detected [104]. The third compares the outputs of two identical threads at different times on two different processors where one thread leads the other in execution by a fixed minimum amount of time [62, 78, 84].

Another approach that also uses a redundant thread running simultaneously also makes use of ECCs to protect the processor's cache memory and some registers. This approach creates redundant threads at runtime for each instruction. Errors are detected when the results of each thread are compared before committing the

results. When an error is detected, the processor "rewinds" to the last committed change, which is immediately prior to the start of the instruction that resulted in the error. This method may also recover from errors by majority voting on the results of redundant threads when there are three or more redundant threads [77].

Another hybrid method combines TMR with ECCs to protect memory [49]. This approach has two layers of protection. ECCs protect each word stored in memory. Each word is also triplicated for added redundancy. The correct output for a specific word stored in memory is the majority vote of the three copies of the word stored in memory.

A different hybrid uses a specialized compiler to determine which instructions in a program are critical to its output at compile time and flags those instructions. A modified super-scalar out-of-order processor detects those flags at runtime and creates one or more replicas of the instruction at runtime. After the processor processes a flagged instruction and its replicas, it compares the results before storing them. If any of the results differ, the processor performs error recovery operations [63].

While ECCs were previously discussed as a hardware redundancy method, they have also been implemented by software in a hybrid approach. This approach assumes that hardware ECCs are unavailable or too expensive for an application. A software program was created that would periodically scrub memory to create codes for words in memory which did not have codes and update codes for words that already had ECC protection. The program allocated space in memory to store the code words since the memory was not designed to accommodate ECCs. The research examined vertical Hamming, vertical cyclic, 2-dimensional, and Reed-Solomon codes and determined that all codes provided a level of protection far greater than unprotected memory, but not as much protection as hardware implemented ECCs [86].

2.3.2.4 Adaptive Redundancy

One method of adaptively responding to changes in error rates chooses an error correcting method that minimizes errors, delays, and energy in cached memory. It utilizes three different software modular redundancy schemes with varying levels of error protection and performance costs. When error rates are low, the lowest cost redundancy scheme with the least error protection is used. As error rates increase, the intermediate cost redundancy scheme with greater error protection is used followed by the one with the highest cost and greatest error protection. As error rates decrease, lower cost redundancy is used [106]. This adaptive method could potentially be applied to TMR, NMR, backup redundancy, hybrid redundancy, or any other redundancy methodology implemented in hardware and/or software to minimize performance costs for a given radiation environment. While Wang et al. apply this adaptive approach to cached memory, this research applies this approach to the processor. The use of adaptive redundancy for a processor has not been extensively explored in the literature before. The only example discovered was previously mentioned in which TMR is turned on and off based on the radiation environment detected by a radiation sensor [55]. This example only provides a choice between redundancy and no redundancy. No examples were found in literature that allowed a choice between different redundancy techniques for a processor.

2.3.3 Summary of Mitigation Techniques

All of the mitigation techniques discussed in this section, with the exception of the TMR adaptive approach, are fixed once implemented on a space vehicle. The TMR adaptive approach is not a viable option since it affords no error protection in the event that a SEU occurs when the processor is in single processor mode. These fixed redundancy methods each come with fixed energy and processing speed advantages

and disadvantages that cannot change for the duration of a space vehicle's mission. This research seeks to answer whether it is possible to implement multiple redundancy methods and switch between them in order to provide a tradespace in terms of energy and processing speed performance.

2.4 Radiation Comparisons

Although making comparisons between different radiation experiments is inherently difficult, even a rough order of magnitude comparison is valuable in the task of evaluating radiation hardening and mitigation efforts. Radiation experiments vary widely in the types of radiation, energy levels, and flux levels used. Experiments that cause SEUs and SETs use particle radiation which are known to cause SEUs and SETs rather than x-rays and gamma rays which are not known to cause SEUs and SETs. Particle experiments may use protons, neutrons, or heavier ions which can be as "light" as helium or as "heavy" as gold (or heavier) in terms of atomic number. Indeed, many elements across the periodic table can and have been used in radiation experiments. In addition, the energy level also plays a major role. The energy level of particles depends on the velocity to which the particles are accelerated before striking their target. If these concerns were not enough, the flux also plays a role. Flux is essentially a particle flow rate or density measure; it describes the number of particles that pass through a cross sectional area over a certain period of time.

In addition to these factors, the materials used to construct the electronic device (i.e. silicon, doping levels, device sizes, and etc.) also have an impact on the rate at which SETs and SEUs occur. As a result, comparing the results of one radiation experiment to another is inherently difficult. Typically, any radiation comparisons are made on a case by case basis and any data on radiation comparisons have come from past empirical data. One method of comparing SEU rates with different radiation

environments is provided by Normand et al. [66, 67]. In these papers, a comparison is made between SEU rates of identical microelectronics exposed to broad spectrum (10 MeV to 1 GeV) neutrons and 400 MeV protons [67] at Los Alamos National Laboratories. The SEU rate comparison was nearly one-to-one across all of the different microelectronic devices that were tested. One additional piece of information from Normand et al. is that the neutrons had a flux of $1 \times 10^9 \frac{n}{cm^2 \cdot hr}$, but the proton flux was unspecified [67]. This neutron flux will be important when making SEU rate comparisons when discussing the results of neutron experiments conducted in this research. Those results are presented in Section 5.2.2.1

While these comparisons are inherently difficult, some similarities may be noted between experiments where similar radiation environments are used. For example, the Cibola flight experiment saw a peak proton flux between 10 and 100 MeV as shown in Figure 1 in [108]. Normand et al. suggested that 200 to 300 MeV protons could also replicate the neutron environment in [66], but did not provide test data to support that claim. Assuming that the Cibola proton environment is similar enough to Normand's proton environment, it could be argued that the Cibola SEU rate could be used to predict the SEU rate in an equivalent neutron environment. This is an unfounded assumption, but making this assumption enables a rough order of magnitude comparison between the Cibola SEU rate and the SEU rate seen in the neutron experiment conducted in this research.

Some other problems with this assumption are the unknown proton fluence levels in Normand's work. Additionally, the goal of Normand's work was to match the SEU rates of microelectronics in a known neutron environment by using proton radiation at varying energy levels using identical electronic devices. Note that the electronic devices used in this research (Intel Cyclone V FPGA on a Terasic DE10-Standard board) differ from the Xilinx FPGAs used by the Cibola flight experiment.

The Cibola flight experiment experienced a mitigated SEU rate of 0.78 SEUs/device/day, or a failure-in-time FIT rate of 3.25×10^7 [108]. This is a good goal for mitigated SEU vulnerability and will serve as a point of comparison for the radiation experiments performed in this research. The Cibola flight experiment's mitigation technique was TMR combined with configuration memory scrubbing.

2.5 Hardware Selection

This research attempts to implement the proposed Adaptive-Hybrid Redundancy (AHR) on an Intel Cyclone V FPGA. Intel FPGAs were chosen because they have not been publicly studied for radiation hardness to the extent that other brands of FPGAs, such as Actel and Xilinx FPGAs, have been studied. This research sought to add to the body of knowledge on radiation hardness of FPGAs by examining an Intel FPGA rather than a more extensively researched brand of FPGA. The Cyclone V was chosen over other available Intel FPGAs because it is representative of an inexpensive Commercial-Off-the-Shelf (COTS) FPGA.

The Cyclone V's unmitigated vulnerability to SEUs and SETs must be understood in order to determine the appropriate rate at which to inject errors into AHR simulations. It is also necessary to determine a baseline with which to compare AHR's performance in the presence of radiation. This will allow a determination to be made about whether AHR is providing mitigation by lowering the number of undetected and uncorrected errors. This research will examine the unmitigated vulnerability of the Cyclone V to configuration and user logic SEUs as well as user logic SETs.

It is expected that the unmitigated Cyclone V will be equally vulnerable as other unmitigated FPGAs; however, this research does not seek to draw conclusions about the radiation vulnerability of various FPGA brands. This is primarily because research on the unmitigated vulnerability of FPGAs has proven difficult to locate.

Much of the existing FPGA radiation research focuses on the error rates of various mitigation strategies when compared against one another and does not consider comparisons to an unmitigated FPGA. Additionally, determining the unmitigated vulnerability of various FPGA brands is out-of-scope for this research. However, as previously noted in the previous section, a rough order of magnitude radiation comparison will be made between a satellite utilizing Xilinx FPGAs and the Cyclone V FPGA used in this research. Because this comparison is an order of magnitude comparison, it is viewed more as a qualitative measure than a quantitative one.

On a final note, this research does not seek to attribute the source of Cyclone V errors to either configuration logic SEUs or user logic SEUs or SETs. This research does seek to establish a baseline of Intel FPGA radiation vulnerability as a secondary objective with the goal of taking the first step toward hardening Intel FPGAs in the same way that other FPGA brands have been hardened.

2.6 Test Approaches

This section discusses some commonly used approaches to determine the effectiveness of mitigation techniques in past works. The test approaches include analysis, simulation, and radiation testing as well as the types of hardware used and the programs implemented in hardware.

Some researchers performed an analysis or proof [48, 52, 59, 87, 91, 95]. Kolla et al. used analysis and mathematical proof to demonstrate that a residue code based error detection method could detect multi-bit errors with 99% accuracy [48]. Lala et al. used a proof to show the effectiveness of their residue code approach to designing self-checking circuits [52]. Mahmoud et al. conduct an analysis to demonstrate the ability of their Triple Modular Redundancy (TMR) voter to mitigate errors [59]. Singh and Gray utilize mathematical analysis to demonstrate the effectiveness of their

redundancy approach which uses TMR and also incorporates replacing failed processors with spares [87]. Sterpone and Du use analysis to demonstrate the effectiveness of their FPGA radiation hardening approach, which utilizes a specialized compiler to add guard gates and other circuitry to mitigate SETs [91]. Tamir uses analysis to demonstrate the effectiveness of his approach to redundancy in a computer network [95].

Many researchers performed simulations to determine radiation vulnerability [3, 23, 47, 49, 55, 57, 71, 73, 78, 82, 91, 92, 93, 101, 106]. Some of the simulations were performed on individual transistors or gates [47]. Other simulations examined combinational and sequential logic such as inverter chains, counters, decoders, and multiplexers [55, 57, 82, 92]. Some simulations were performed on memory [3, 49, 73, 93]. Other simulations were performed on processors running benchmark programs: some of these simulations used hardware-in-the-loop (HITL) [23, 71, 78, 79, 91, 94, 101, 106]. Many of the simulations also injected faults to determine how their redundancy schemes would detect and correct the errors. Fault injection was performed with software for software only simulations. Fault injection was performed using software manipulation or direct electrical injection for HITL simulations [23, 57, 71, 73, 92, 94, 106].

Other researchers performed radiation testing to determine radiation hardness or vulnerability [3, 31, 43, 57, 73, 82, 88]. These researchers used a number of different types of ions including protons, neon, argon, cobalt, copper, krypton, silver, and xenon [3, 43, 57, 82, 88]. Some of the researchers performed radiation tests on memory [3, 43]. Rezgui et al. performed tests on inverter chains and other combinational and sequential logic structures [82]. Ostler et al. implemented shift registers [73]. Other researchers performed their tests on processors running programs during the tests [31, 57, 88].

Another common approach to determining vulnerability of a device is to implement a static pattern. The device is then irradiated and the pattern on the device is compared to the original pattern. This comparison can be performed during or after irradiation. When the two patterns differ, an error has occurred. Common patterns include all 0's, all 1's, and checkerboard patterns [9, 25, 28, 43, 61].

This research presents a combination of analysis, software simulation, and HITL simulation to determine the effectiveness of AHR and using radiation testing on a checkerboard pattern to determine the vulnerability of the Intel Cyclone V FPGA.

2.7 Background Summary

This chapter discussed Single Event Effects (SEEs) and focused on the impact of Single Event Upsets (SEUs) and Single Event Transients (SETs) because mitigating the effects of SEUs and SETs through the use of Adaptive-Hybrid Redundancy is a focus area of this research. Additionally, numerous redundancy techniques were discussed. Of these, the simplest Triple Modular Redundancy (TMR) and Temporal Software Redundancy (TSR) strategies were selected to implement AHR.

The chosen TMR strategy is system level TMR that triplicates a processor, utilizes a majority voter to determine the correct output, detects single processor errors and corrects them by copying the state of the agreeing processors to the processor that disagrees, and corrects multiple processor errors by restoring all three processors to a previously saved state. This strategy was chosen because it is the simplest of the TMR approaches and it is typically used for U.S. Air Force space vehicles.

The chosen TSR strategy is Error Detection by Duplicated Instructions (EDDI), which duplicates all instructions, adds check instructions before all store word instructions, adds code to create save/restore points which store the system state, and adds error recovery code to recover to a save/restore point when an error is detected at

one of the check instructions. This TSR strategy was chosen because of its simplicity and to ensure that more complicated TSR strategies would not obscure the results of this research such that it would be difficult to determine the advantages of AHR over TMR or TSR alone. This choice also reduces the scope of work because choosing to implement a more complex TSR method would substantially increase the workload. A simpler TSR method enables this research to focus more exclusively upon AHR.

This chapter also discussed radiation comparisons so that a useful comparison could be made between SEU rates from a real space mission and the the SEU rates measured from neutron radiation testing performed on an Intel Cyclone V FPGA in this research. These comparisons enable a determination of the appropriate SEU rate to use in simulations of TMR, TSR, and AHR.

Finally, this chapter looked at previous testing and analysis approaches used by other researchers looking at SEU mitigation through redundancy and it was determined that a simulation and analysis approach would be used for this research to demonstrate the advantages of AHR over TMR and TSR alone.

III. AHR MIPS Development

3.1 Introduction

In order to answer the research questions posed in Section 1.3, it is necessary to develop a processor architecture to implement Adaptive-Hybrid Redundancy (AHR). This architecture should incorporate multiple redundancy methods and switch between these methods. The architecture would therefore be adaptive in that it could switch between two or more redundancy methods. The architecture would also be hybrid to allow for two or more different types of redundancy, and more specifically, a combination of hardware and software redundancy. The selected hardware approach is a form of Triple Modular Redundancy (TMR) while the selected software approach is a form of Temporal Software Redundancy (TSR). The adaptive nature of the architecture will allow it to switch between TMR and TSR depending on the energy and performance requirements for the processor at any given time.

The TMR approach implemented in this research is a system level approach where three copies of the same processor operate on the same inputs and their outputs are examined by a voting circuit. All three processors produce the same outputs when there are no errors. If only one of the three processors encounter an error, the correct output can be determined by majority vote. In the case of a single processor's output differing from the other two, the voter circuity resets the "incorrect" processor and sets the processor's internal state to match the internal state of the two "correct" processors. If all three processors disagree on the output (i.e. no majority exists), the voter resets all three processors and restores them to a previously saved internal state. This approach also assumes that the save/restore point is saved in radiation hardened memory [6, 7, 10]. This approach was presented as the first TMR approach in Section 2.3.2.1.

The TSR approach selected for this work is one in which all instructions except for store instructions are duplicated. A branch instruction is inserted before all store instructions to ensure that the value to be stored to memory and its duplicate are equal. This was the first TSR example provided in Section 2.3.2.2 and was illustrated in Table 1. This TSR approach is called Error Detection by Duplicated Instructions (EDDI) [71].

The AHR architecture builds upon these TMR and TSR approaches by combining them in a common architecture with a controller selecting between the two. The intent is that the TMR architecture could be fully utilized when the TMR mitigation strategy was needed for faster processing and the TSR architecture could be used when energy conservation is important. The TSR program will run on one of the TMR architecture's three processors while bypassing the TMR Voter, thereby avoiding a fourth processor. To experiment with such an approach, this architecture will be implemented on an FPGA due to the ease and speed of implementation on an FPGA by encoding the architecture in VHDL. For this work, the architecture was specifically implemented on a Terasic DE10-Standard board with an Intel Cyclone V FPGA.

The AHR architecture requires three identical processors for TMR and the use of one of these processors for TSR. The experimental processor should be designed with testing in mind. Testing requires comparing TMR and TSR performance in the presence and absence of errors. Testing must be able to inject errors and observe the effects in both TMR and TSR; namely testing will determine how TMR and TSR respond to errors and the performance penalty (time to complete a program and energy consumed in completing the program) of responding to those errors. A simplified MIPS processor, called Basic MIPS, is designed and used to fulfill these conditions.

The design of the Basic MIPS processor and its associated programs are detailed in

Section 3.2. The Basic MIPS processor then facilitates the TMR MIPS architecture and program development in Section 3.3. Basic MIPS development also provides insight into the development of the TSR architecture and programs in Section 3.4. After TMR MIPS and TSR MIPS are fully understood, the discussion will return to the development of AHR in Section 3.5.

3.2 Basic MIPS

3.2.1 Basic MIPS Development

Basic MIPS is an architecture that implements 33 instructions from MIPS32™. These instructions are shown in Table 2. It was designed in such a way that the two fundamental building blocks are a 2-input NAND gate and a D-Flip-Flop (also called a register). All other components are comprised of these two building blocks, up to and including the Controller and Datapath which comprise the two interconnected components of the Basic MIPS processor. There is no additional logic required by the Basic MIPS processor outside of the Controller and Datapath. This design strategy allowed for complete insight into the operation of the processor and full traceability of all injected errors. If a previously designed processor were used, significant time and resources would have been taken away from implementing AHR to understanding the processor and adapting it to the error injection and analysis requirements as well as ensuring it is compatible for use in TMR, TSR, and AHR.

The only storage elements this architecture contains are a 30-bit register to store the program counter (PC) in the Controller; a 32-bit register to store an instruction in the Controller; a 4-bit register to store the state of the Controller; thirty two 32-bit registers of which 31 are user programmable while one is always a 32-bit zero in the Datapath; and a few single bit registers in the Datapath to account for timing delays when processing certain instructions. Instructions to be run on Basic MIPS are

stored separately in memory. As previously mentioned in the assumptions in Chapter I, memory is immune to errors and not a part of the processor. These instructions are retrieved from memory and operated upon sequentially; each instruction is retrieved and operated upon until completion before the next instruction is retrieved.

Before continuing to a discussion on Basic MIPS interface with memory, it is important to note why a sequential design was chosen over a pipelined or super-scalar design. A sequential processor is far simpler and less time-consuming to implement than a pipelined or super-scalar processor. Pipelined and super-scalar processors are also irrelevant to implementing AHR which switches between TMR and TSR since pipelined and super-scalar processors would simply speed up both TMR and TSR.

Table 2. List of Implemented Basic MIPS Instructions

Abbreviation	**Name**
SLL	Shift Word Left Logical
NOP	No Operation
SRL	Shift Word Right Logical
SRA	Shift Word Right Arithmetic
SLLV	Shift Word Left Logical Variable
SRLV	Shift Word Right Logical Variable
SRAV	Shift Word Right Arithmetic Variable
ADD	Add Word
ADDU	Add Unsigned Word
SUB	Subtract Word
SUBU	Subtract Word Unsigned
AND	Bitwise Logical And
OR	Bitwise Logical Or
XOR	Bitwise Logical Exclusive Or
NOR	Bitwise Logical Not Or
SLT	Set on Less Than
SLTU	Set On Less Than Unsigned
BGEZ	Branch on Greater Than or Equal to Zero
BLTZ	Branch on Less Than Zero
BEQ	Branch on Equal

Table 2 – *Continued on next page*

Table 2 – *Continued from previous page*

Abbreviation	**Name**
BNE	Branch on Not Equal
BLEZ	Branch on Less Than or Equal to Zero
BGTZ	Branch on Greater Than Zero
ADDI	Add Immediate Word
ADDIU	Add Immediate Unsigned Word
SLTI	Set On Less Than Immediate
SLTIU	Set On Less Than Immediate Unsigned
ANDI	Bitwise Logical And Immediate
ORI	Bitwise Logical Or Immediate
XORI	Bitwise Logical Exclusive Or Immediate
LUI	Load Upper Immediate
LW	Load Word
SW	Store Word

Basic MIPS communicates with memory using six signals. These signals are illustrated in a simple block diagram shown in Figure 3. The inputs are on the left and begin with "i_" and outputs are on the right and begin with "o_". The signal "i_clk" is the clock signal and "i_reset" is used to reset Basic MIPS so that all registers are set to zero. Basic MIPS requests data from memory by asserting the "o_MEM_READ" signal and providing an address on the "o_MEM_ADDRESS" signal. Memory responds by providing the data at the given address on "i_MEM_OUT" and asserting the "i_MEM_READY" signal. Basic MIPS stores data to memory by providing an address on the "o_MEM_ADDRESS" signal, the data to store on the "o_MEM_IN" signal, and asserting the "o_MEM_WRITE" signal. Memory responds by asserting the "i_MEM_READY" signal to indicate that the data provided was stored to the address supplied by Basic MIPS.

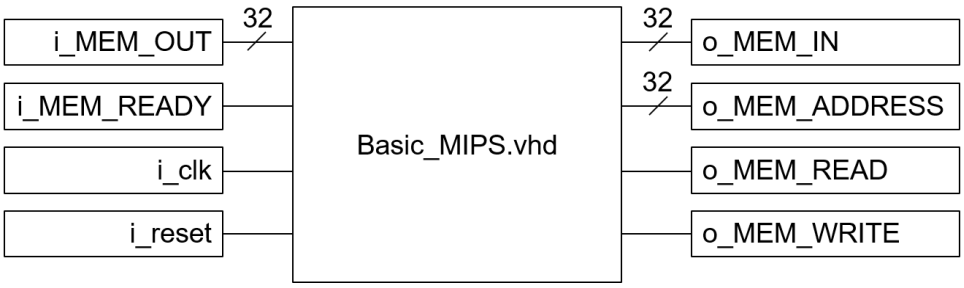

Figure 3. Basic MIPS Inputs and Outputs

A detailed Air Force Institute of Technology technical report on the Basic MIPS Architecture covers all of Basic MIPS' inner workings and can be made available upon request [36].

3.2.2 Basic MIPS Programs

Because the Basic MIPS architecture does not support all MIPS instructions, no existing benchmarks can be used to evaluate the performance of the Basic MIPS processor. Therefore, programs are generated to evaluate its performance as a baseline against which the performance of TMR, TSR, and AHR may be compared. The processor's performance is evaluated in terms of time and energy required to complete a program. The programs are randomly generated to ensure that all Basic MIPS instructions are fairly represented across multiple programs. The distribution of instructions is intended to be uniform, but that is not necessarily the case after all processing actions to create programs are completed.

A set of Basic MIPS instructions is first created with variable names instead of register names. The set of instructions is randomly generated using all of the Basic MIPS instructions except the branching instructions and load word instruction. The first instruction is required to be a load upper immediate instruction with a uniformly random immediate value. After the first instruction, arguments to the following instructions must come from the zero variable (analogous to the 0 register in the

Basic MIPS architecture in that its value is always 0) or a randomly selected variable that was the result of a previous instruction. In order to simplify things, store word instructions always have the 0 variable as their base. The offset portion of store word instructions are not randomized, and with a base address of 0, the offset specifies the desired memory location. Additionally, the immediate value of any immediate function is always uniformly random. Finally, the last instruction must be a store word.

This randomized set of instructions, however, may not be representative of real world programs which are hardly random. Real programs are carefully designed and implemented by computer programmers, computer scientists, and software engineers. This random approach creates a lot of variables that are never used or stored to memory. Real world programs do not typically create unused variables, so the next step is to eliminate all instructions that produce unused variables from the set of Basic MIPS instructions. All variables that contribute to the computation of a variable which is stored to memory by a store word instruction are considered essential while those that do not are considered superfluous and removed by this step. Variables which are stored to memory by a store word instruction at this step are considered to be the desired end results of the program and are stored to permanent memory.

After eliminating all unused variables, variable names are converted to register names. However, Basic MIPS only has 31 user definable registers. Additionally, one more register is reserved for a loop counter for reasons that will be discussed in a few paragraphs. This means that the set of Basic MIPS instructions may never use more than 30 registers. If the variable instruction set never uses more than 30 variables, then the transformation process is very simple and each variable can be assigned to a single register. If more than 30 variables are used, they can be carefully examined to determine if more than 30 variables are in use simultaneously. For example, a program

might use 40 variables, but some may only be used for a short period of time before storing them to memory and never using them again: this could occur in such a way that there are no more than 30 variables being used at any given time. Registers holding variables that have been permanently stored to memory and never used again are freed for use by other variables. In this scenario, it is also fairly simple to assign variables to registers so long as variable names to register assignment mappings are carefully tracked. If more than 30 variables need to be used simultaneously, then extra steps must be taken in the variable to register assignment process and extra instructions must be added. If 30 variables are currently being used and the next instruction is creating a new variable that must be assigned a register, an algorithm determines which of the 30 variables currently in registers is going to be accessed the furthest amount of time from the next instruction. The algorithm then inserts a store word instruction to store that variable to a temporary storage location in memory, then the new variable is assigned to the register that was just vacated. When the variable stored to temporary storage in memory is needed, it is reloaded into a register. If all the registers are full when this reload must occur, another variable is selected (again the variable that is going to be accessed the furthest amount of time from the next instruction) to be temporarily stored to memory to make space for the reload. All of these determinations are made at compile time by the compiler (a series of Matlab functions) which does the variable to register mapping.

A second DE10-Standard's Cyclone V was used to store the set of instructions rather than in the DE-10 Standard's Static Random Access Memory (SRAM) in order to isolate the memory from radiation effects and enable energy measurements of the processor alone. The set of instructions were implemented in a memory emulator or "Memulator" This Memulator stores all of the instructions in a series of 32-bit registers on the Cyclone V. This was accomplished by using VHDL code to create

the Memulator as well as provide the interface for Basic MIPS to read and write from the Memulator.

Storing the set of instructions in a Memulator on a Cyclone V greatly restricted its size. The Cyclone V only possesses 166,036 registers capable of holding no more than 5,188 32-bit memory locations. It is also limited by routing constraints to read and write to all of the registers, which means that not all of the 5,188 addresses are usable. The limited size meant that running the set of instructions one time in hardware would make the runtime very short. This in turn limits the observability of the duration and energy usage when the FPGA implementing the Basic MIPS processor runs the set of instructions. To overcome this observability limitation, additional code was added to create a loop around the set of instructions. This loop would cause the set of instructions to execute 999 times to increase the observability of the time to complete the 999 loops through and the energy used by the processor. In order to add this loop, one of the 31 user registers was reserved for the loop counter (counts from 999 down to 1) and no longer available as a user definable register. That is why the variables must be mapped to 30 user registers.

After the end of the set of instructions and loop code, empty memory locations are added (containing all 0s) for use as temporary and permanent memory. As previously mentioned, temporary memory is used to store variables when there are more variables than registers available while permanent memory is used to store program results. The instruction set, loop code, and temporary and permanent memory are collectively referred to as a program.

A sample Basic MIPS program is shown in Table 3 without the temporary and permanent memory. In this example, the first two instructions set up the loop counter. The first instruction loads 999 into the upper 16 bits of R31 so that the value stored in R31 is $999 \cdot 2^{16}$. The second instruction shifts the value 16-bits to the right, effectively

dividing by 2^{16} and changing the value in R31 to 999. The third instruction marks the beginning of the randomly generated set of instructions. Most of the instructions are omitted here for brevity, but the final store word instruction is shown on line 46. The next instruction subtracts one from the loop counter. Instruction 48 branches back to the start of the loop on line 3 if the value in R31 is greater than 0. Once the loop counter reaches 0, the branch on line 48 is not taken and code execution proceeds to line 49. This instruction causes code execution to branch to a region outside the program. When the processor attempts to read the instruction from the Memulator, the Memulator recognizes that the instruction is outside the range of addresses allocated for the program. The Memulator further understands that this means the program is complete.

Table 3. Basic MIPS Code Example

Line Number	**Instruction**
1.	LUI R31 999
2.	SRL R31 R31 16
3.	LUI R1 30277
4 - 45	...
46.	SW R24 R0 228
47.	ADDI R31 R31 -1
48.	BGTZ R31 -45
49.	BEQ R0 R0 END

Upon program completion, the Memulator sends a "DONE" signal to reset itself and the Basic MIPS processor. This causes the program to start over from line 1 in the example above.

One additional constraint on the program is that it must have at least enough instructions that a TMR to TSR transition is triggered in AHR. If a program running AHR completes in TMR mode, AHR will never transition from TMR to TSR. Since

the main purpose of this research is to develop AHR and evaluate its performance compared to TMR and TSR, AHR must transition from TMR to TSR. Therefore, any program which has insufficient instructions to trigger the TMR to TSR transition is discarded.

3.3 TMR MIPS

3.3.1 TMR MIPS Development

To implement TMR, three Basic MIPS processors are connected to a voter which is then connected to memory as shown in Figure 4. This figure also combines memory signals into directional arrows and a bi-directional arrow is used to denote the two-way flow of communication between the voter and memory. Notice how there is no direct connection between the Basic MIPS processors and memory. The Basic MIPS processors each communicate with the voter as if they were communicating with memory using the exact same control signals as Basic MIPS normally uses to communicate with memory. The voter compares the memory control signals "o_MEM_READ", "o_MEM_WRITE", "o_MEM_ADDRESS", and "o_MEM_IN" to determine if all three processors are working in unison. During normal operation, if all three processors are attempting to read data from memory, all three processors will simultaneously signal a read operation and provide an address. The voter will confirm that all three processors are attempting to read from the same address, then the voter will attempt to read data from the supplied address in memory. Memory then provides the data and ready signal to the voter. The voter then simultaneously provides the data and a ready signal to each of the Basic MIPS processors. For a write operation, all three processors simultaneously signal a write operation and provide data and an address. The voter confirms that all three processors are attempting to write the same data to the same address, then the voter attempts to write the data to the supplied address

in memory. Memory then provides the ready signal to the voter. The voter then provides the ready signal to each of the processors.

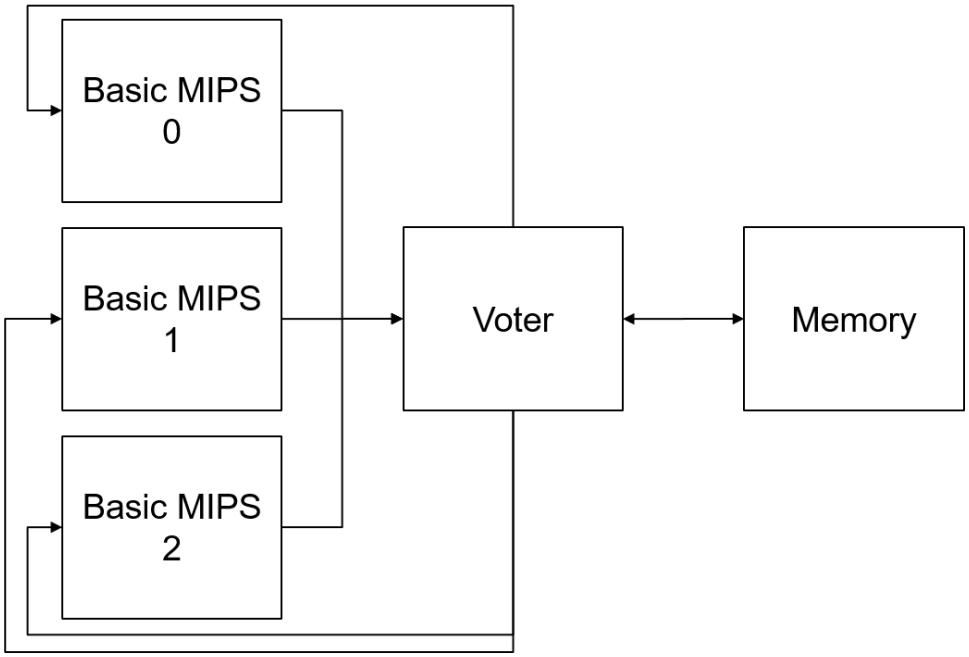

Figure 4. TMR MIPS Block Diagram

If one of the three processors suffers an error such that it fails to provide a read or write signal at the same time as the other two processors, the address provided is different than the other two processors, or the data provided for a write operation differs from the other two processors' provided data, a single processor error is detected. This error is called a TMR Type A error. When this occurs, the voter switches into a single processor error state. The voter first resets the processor that is in error and stores the PC value of the two agreeing processors at which the error occurred. Then, a store word command is issued to the two agreeing processors to store the data in user defined register 1 (R1). The two agreeing processors will then attempt to store R1 to memory, which is really the voter. The voter takes the value of R1 from the two agreeing processors and confirms that the two R1 values are equal. Then, the voter issues a load word command to the processor in error to load a value into R1. When

the processor in error attempts to load a word from memory, the voter responds by supplying the R1 value from the two agreeing processors. This process is repeated for the remaining 30 user registers (R2-R31, not R0 because it is the zero register). At this point, all of the user registers in the processor in error now match those in the agreeing processors. However, all three processors PC values do not match the value at which the error was detected. The voter then issues branch commands to all three processors to return them to the PC value at the point where the error was detected. This process is shown as a flow chart in Figure 5. This figure references a Type B error which will be discussed shortly. When a Type B error occurs, Type B error recovery begins.

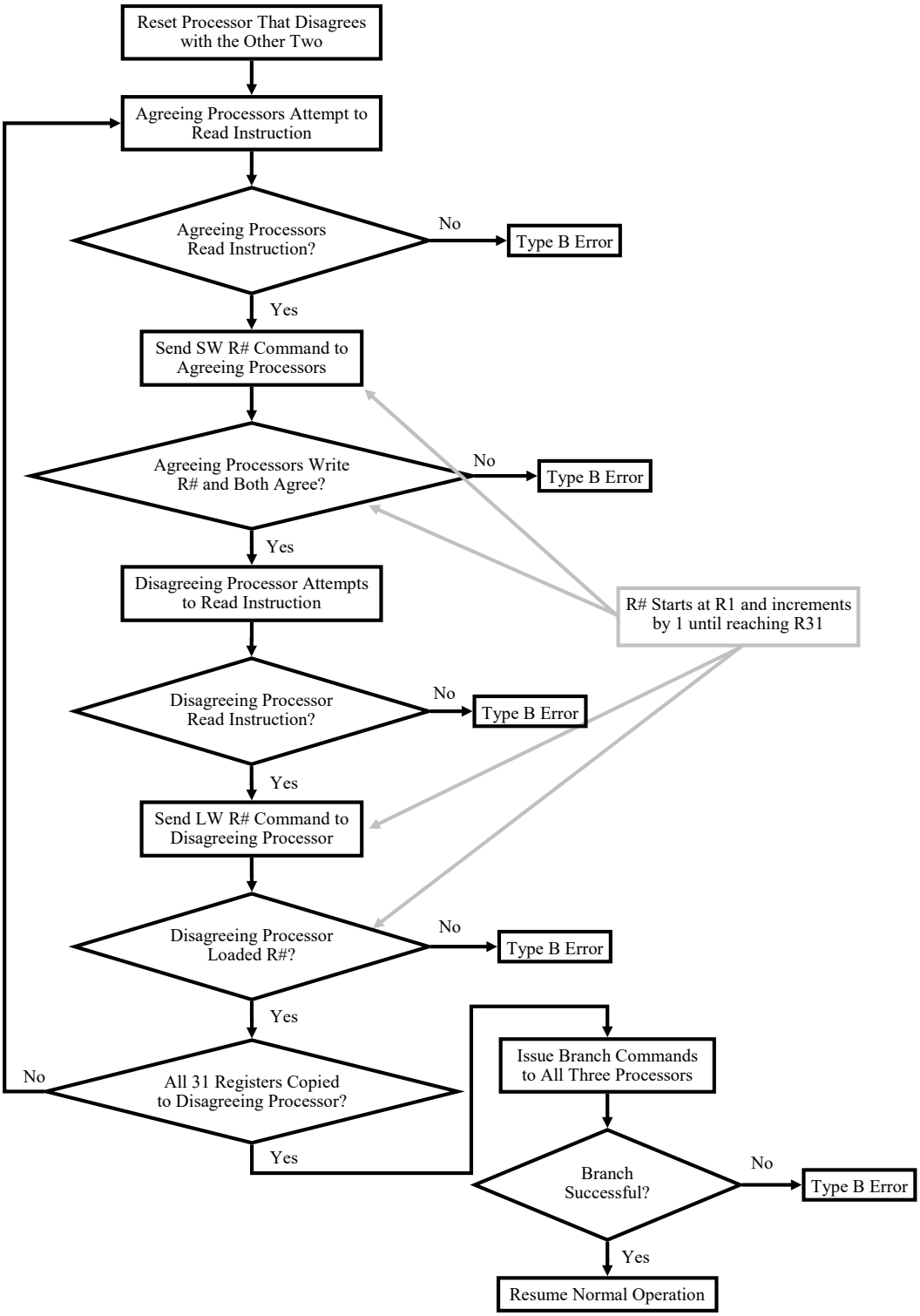

Figure 5. TMR MIPS Type A Error Recovery Flow Chart

If two or three processors suffer an error such that they do not agree on the type of operation, read or write, the address, or the data to be written for a write operation, then a multiple processor error has occurred. This error is called a TMR Type B error. It is possible that one of the three processors is correct, but the voter has no way to determine which one is correct. The voter resets all three processors, and begins performing error recovery operations to restore the internal state of the processors to a previous save/restore point. Processing resumes from that save/restore point and will eventually return to the location in the program at which the error occurred, then proceed past that point so long as another multiple processor error does not occur.

The save/restore point consists of 65 memory locations split into two main segments and one additional memory location bit. Each of the two main segments can store the value of the 30 user definable registers, the loop counter from R31, and the PC at a specific moment in time. The additional memory location stores a value the indicates which of the two segments contains the most recently created save/restore point.

During save/restore point creation, the TMR Voter determines which segment was most recently updated. The TMR Voter then instructs the Basic MIPS processors to store R1 to the first location in the segment that was not most recently updated. The TMR Voter ensures that at least two of the three R1 values match, then writes it to memory. The TMR Voter continues this process until all 30 user definable registers, the loop counter in R31, and the PC are written to memory. Finally, the TMR Voter updates the additional memory location bit to point to the new most recently updated segment. This processes is illustrated as a flow chart in Figure 6.

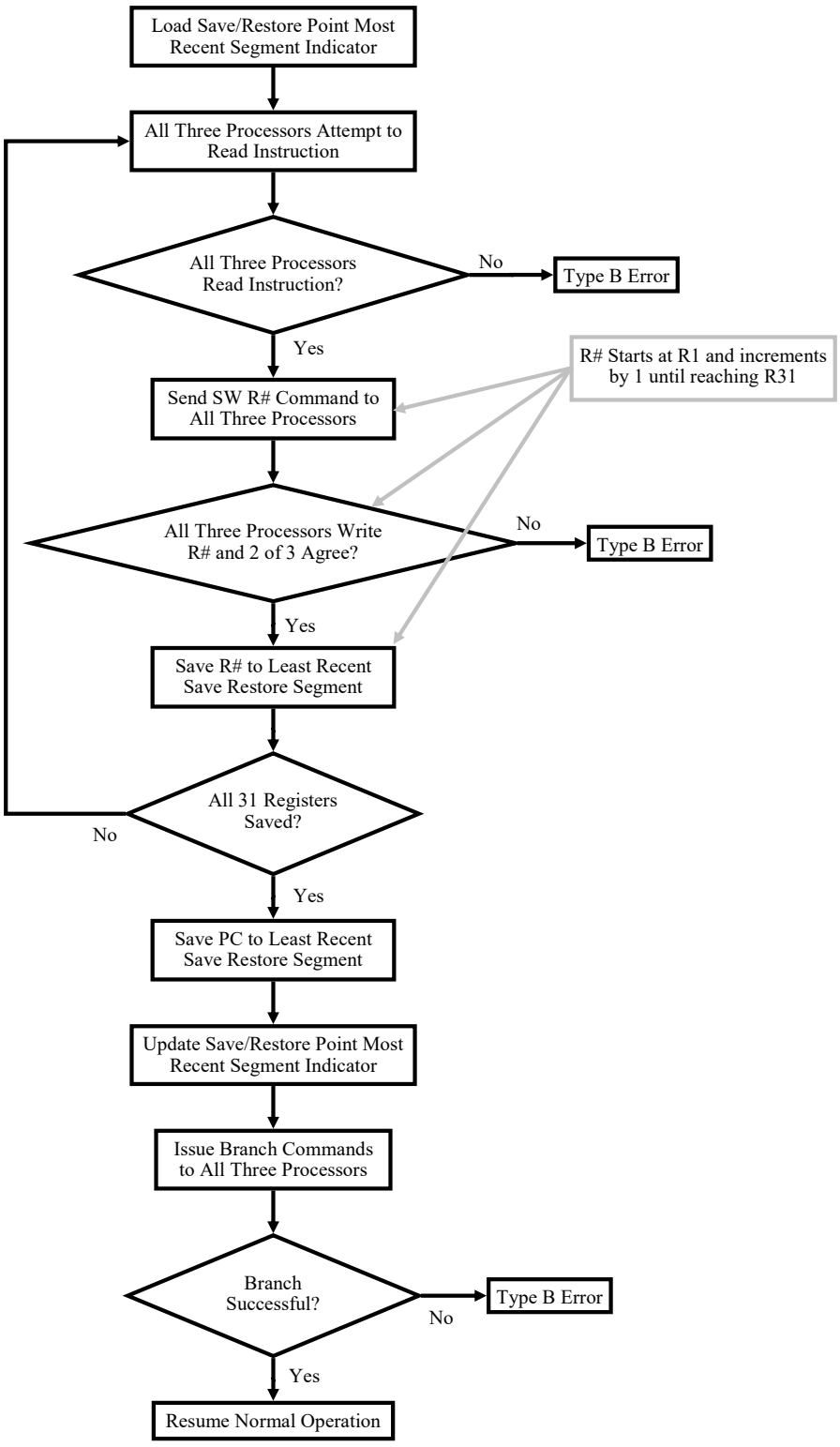

Figure 6. TMR MIPS Save/Restore Point Creation Flow Chart

The TMR Voter creates save/restore points after TMR MIPS processes a predetermined number of instructions. The TMR Voter does not possess the capability to determine when a program is starting or ending a loop; therefore, the TMR Voter cannot intentionally create save/restore points based on progress through the loop or the value of the loop counter. This is why the save/restore point must store the PC value as well as all the register values. One constant that must be defined for these experiments is the number of instructions between save/restore point creation for TMR MIPS. This value was arbitrarily set to be 10,000. The number of save/restore points created by TMR MIPS, will depend upon the total number of instructions in the program. For example, a program with only 12,000 instructions (including all 999 loop iterations) will only have one save/restore point created while a program with 95,000 instructions will have nine save/restore points created.

During Type B error recovery operations, the TMR Voter first resets all three processors, then issues load word commands to all three processors to sequentially load R1 to R31 from the most recent save/restore point in memory. Once R1 to R31 have been restored from memory in all three processors, all three processors are issued a branch command to return them to the PC value at which the most recent save/restore point was created. Figure 7 illustrates Type B error recovery as a flow chart.

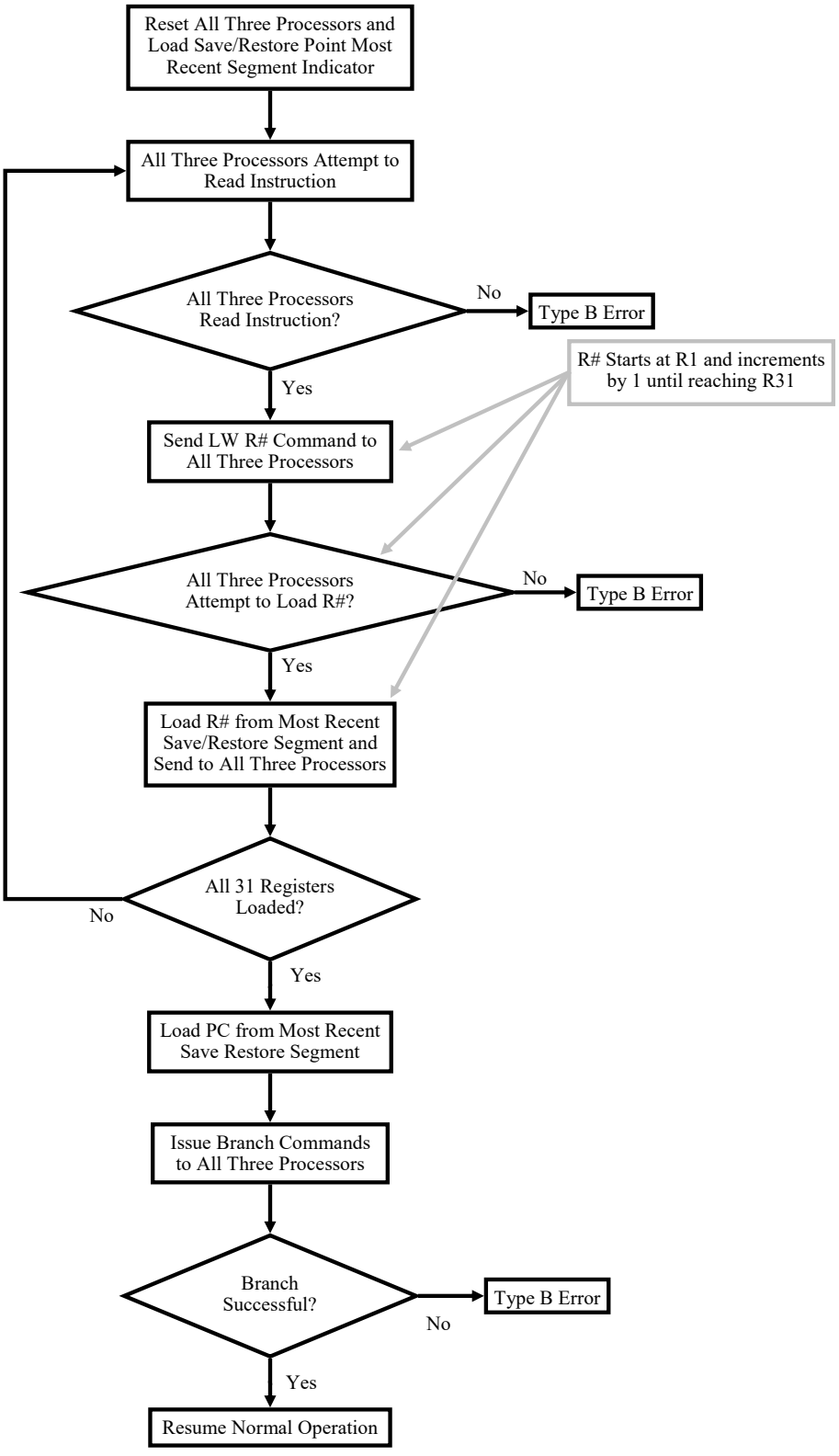

Figure 7. TMR MIPS Type B Error Recovery Flow Chart

A Type B error can also be triggered when all three processors fail to interact with memory before a predefined timeout period. This indicates that the processors are stuck in a state from which they cannot recover without voter intervention. A Type B error is also triggered when recovering from a Type A error if the two processors that were thought to be in perfect agreement disagree on the value of a register or stop responding to the voter. A Type B error also occurs if all three processors disagree on the value to be stored during save/restore point creation.

One final note is that it is assumed that the TMR Voter is hardened such that it is immune to errors. If the TMR Voter were subjected to SEUs and SETs, its internal state could be affected by radiation with adverse consequences to the behavior of TMR MIPS. A state error could place the TMR Voter in conflict with the Basic MIPS processors in such a way that the TMR MIPS processor would freeze. A state error could also trigger error recovery when no error has occurred or save/restore point creation when it is not time to create a save/restore point. Another possibility is that error recovery or save/restore point creation could skip steps leaving some portions of the processor vulnerable to undetectable errors.

A detailed Air Force Institute of Technology technical report on the TMR MIPS Architecture covers all of TMR MIPS' inner workings and can be made available upon request [38].

3.3.2 TMR MIPS Programs

TMR MIPS programs are identical to Basic MIPS programs and are used by TMR MIPS with minimal modification. The only modification is the addition of 65 blank memory locations to be used by the save/restore point.

3.4 TSR MIPS

3.4.1 TSR MIPS Development

The TSR MIPS architecture is identical to Basic MIPS. The TSR MIPS processor is the Basic MIPS processor. The only difference is that TSR MIPS makes use of a specialized program that incorporates redundancy while still computing the same results as an equivalent Basic MIPS program.

3.4.2 TSR MIPS Programs

TSR MIPS program generation follows the same process for register assignment as Basic MIPS programs in Section 3.2.2; however, there are far more store and load instructions required for managing temporarily stored variables because TSR can only access 14 user definable registers instead of 30. TSR only has access to 14 user definable registers because 2 registers are reserved for the counter (leaving 29 registers) and every register must have a duplicate (29/2 = 14.5). This leaves 1 unused register. One final note is that any store word instructions that were part of the Basic MIPS program before the addition of store instructions to temporarily store variables to memory are also in the TSR MIPS program. These original store word instructions are viewed as storing words to a permanent memory which could be utilized later. The values stored in permanent memory will be identical for Basic MIPS, TMR MIPS, and TSR MIPS while the values in temporary memory will not be identical.

Each TSR program also contains code to create a loop around the set of instructions as was done for Basic MIPS programs. This loop also executes the set of instructions 999 times. At the end of the set of instructions, and before the branch to restart the loop, additional instructions are added to determine when save/restore point creation instructions should be executed. A save/restore point is created when

the counter equals 750, 500, and 250. After the end of the set of instructions and loop code, empty memory locations are added (containing all 0s) for use as temporary and permanent memory.

Table 4 shows a sample TSR MIPS program without the temporary and permanent memory. Also excluded are save/restore point creation instructions, error recovery instructions, and save/restore point memory which will be further discussed in the following paragraphs.

In the example in Table 4, the first four instructions set up the loop counter and its duplicate. The first two instructions load 999 into the upper 16 bits of R30 and R31 so that the values stored in R30 and R31 are both $999 \cdot 2^{16}$. The third and fourth instructions shift the values in R30 and R31 16-bits to the right, effectively dividing by 2^{16} and changing the value in R30 and R31 to 999. The fifth instruction marks the beginning of the randomly generated set of instructions and the sixth instruction is the fifth instruction's duplicate. Most of the instructions are omitted here for brevity, but the final store word instruction is shown on line 92. Note the branch instruction on line 91 that ensures that R12 and its duplicate, R26, are equal before proceeding to the store word instruction. If they are not equal, code execution jumps to the error recovery code which happens to be 129 addresses away from the branch instruction. The next two instructions compare the current counter value to 250. If the loop counter value equals 250, then code execution jumps to the save/restore point creation instructions which happen to be 19 addresses away from the branch instruction on line 94. Similarly, the instructions on lines 95 and 96 determine if the counter equals 500 and lines 97 and 98 determine if the counter equals 750. In both of these cases, if the counter equals either of these values, code execution also jumps to the save/restore point creation instructions. If the counter does not equal 250, 500, or 750, code execution proceeds to instructions 99 and 100 which subtract one

from the loop counter stored in registers R30 and R31 respectively. The values in R30 and R31 are compared to ensure the loop counter is not in error before proceeding to the branch instruction on line 102. If the values are not equal, code execution jumps to error recovery instructions which happens to be 119 addresses away from the branch instruction on line 101. If the values match and are greater than 0, the branch instruction on line 102 causes code execution to return to the beginning of the loop on line 5. If the loop counter value is 0, code execution proceeds to line 103. This instruction causes code execution to branch to a region outside the program. When the processor attempts to read the instruction from the Memulator, the Memulator recognizes that the instruction is outside the range of addresses allocated for the program. The Memulator further understands that this means the program is complete and sends the DONE signal to reset itself and the Basic MIPS processor running the TSR instructions.

Also note that the example in Table 4 is the TSR equivalent program to the Basic MIPS program example in Table 3.

Table 4. TSR MIPS Code Example

Line Number	**Instruction**
1.	LUI R30 3
2.	LUI R31 3
3.	SRL R30 R30 16
4.	SRL R31 R31 16
5.	LUI R1 30277
6.	LUI R15 30277
7 - 90	...
91.	BNE R12 R26 129
92.	SW R12 R0 444
93.	ADDI R29 R30 -250
94.	BEQ R29 R0 19
95.	ADDI R29 R30 -500

Table 4 – *Continued on next page*

Table 4 – *Continued from previous page*

Line Number	**Instruction**
96.	BEQ R29 R0 17
97.	ADDI R29 R30 -750
98.	BEQ R29 R0 15
99.	ADDI R30 R30 -1
100.	ADDI R31 R31 -1
101.	BNE R30 R31 119
102.	BGTZ R31 -97
103.	BEQ R0 R0 224

TSR MIPS also requires save/restore point creation instructions that create a save/restore point in memory and error recovery instructions to restore TSR MIPS to a previously saved state if an error occurs. Unfortunately, while many previous works discussed save/restore points and branching to error recovery instructions upon encountering an error, no previous works were located that explained these processes [68, 75, 71, 79, 95, 107]. This research developed these processes, but it is unknown if they are unique. Before examining how save/restore points are created and recovery operations are performed, a look at the structure of the save/restore point is needed.

The save/restore point is a block of memory reserved in order to restore the internal state of TSR MIPS. This memory consists of two main segments and one additional memory location. The two main segments each store the 14 user variables stored in the user defined registers at some point in time. Each segment also stores the loop counter value at that same point in time. Of the two main segments, one will have been saved more recently, and the other less recently. The additional memory location stores a value used by the TSR MIPS save/restore point creation code and error recovery code to show which of the two segments is the more recent one.

The save/restore point creation code saves the current values of the 14 user defined registers and the loop counter to the less recently updated segment, then updates the

additional memory location to indicate that this segment is now the more recently updated segment. Before each register value is stored to memory, the code compares duplicated registers. If they match, the code stores them to memory. If they do not match, error recovery is used to restore TSR MIPS to the more recent save/restore point. The additional memory location is only updated if all of the registers match. At this point, the less recently updated save/restore point now becomes the more recently updated save/restore point. After the additional memory location is updated, a branch command is issued to return code execution back to the end of the TSR program where the loop counter will be decreased before branching back to the beginning of the loop.

Table 5 shows a sample of the TSR save/restore point creation instructions. A portion of the instructions have been omitted for brevity. In this particular example, 1300 is the memory location of the save/restore creation point additional memory location that indicates which segment is the most recent one. After that value is loaded, it is compared to 0 and code execution jumps by 3 instructions if it is greater than 0, or continues to the next instruction if it is less than or equal to 0. This value only takes on the values of 0 or 60. The value 0 indicates that the first segment is the most recently updated while 60 indicates that the second segment is the most recently updated. If the value is 0, the next instruction changes it to 60, then branches 2 more instructions forward, If the value is 60, the branch jumps 3 instructions forward, then changes its value to 0. Both paths lead to the BNE R1 R15 102 instruction on line 6. This is the comparison that ensures R1 and its duplicate R15 are a match, if not, code execution jumps to the error recovery instructions, which happens to be 102 memory locations after line 6. These same instructions are repeated for each of the remaining user defined registers and also the loop counter. The instructions for the loop counter are seen on lines 211 to 217.

Notice that if the most recently created save/restore point is in the first segment, it takes six instructions to save the value of a register to the second segment. If the most recently created save/restore point is in the second segment, it takes only five instructions to save the value of a register to the first segment. This means that it will take more instructions and time to create a save/restore point in the second segment than in the first segment. The exact amount of time to create a save/restore point in the first and second segments will be provided in Section 4.4.

Table 5. TSR MIPS save/restore Point Creation Instructions Example

Line Number	**Instruction**
113.	LW R29 R0 1300
114.	BGTZ R29 3
115.	ADDI R29 R0 60
116.	BEQ R0 R0 2
117.	ADDI R29 R0 0
118.	BNE R1 R15 102
119.	SW R1 R29 1180
120.	LW R29 R0 1300
121.	BGTZ R29 3
122.	ADDI R29 R0 60
123.	BEQ R0 R0 2
124.	ADDI R29 R0 0
125.	BNE R2 R16 95
126.	SW R2 R29 1184
127 - 203	...
204.	LW R29 R0 1300
205.	BGTZ R29 3
206.	ADDI R29 R0 60
207.	BEQ R0 R0 2
208.	ADDI R29 R0 0
209.	BNE R14 R28 11
210.	SW R14 R29 1232
211.	LW R29 R0 1300
212.	BGTZ R29 3
213.	ADDI R29 R0 60

Table 5 – *Continued on next page*

Table 5 – *Continued from previous page*

Line Number	Instruction
214.	BEQ R0 R0 2
215.	ADDI R29 R0 0
216.	BNE R30 R31 4
217.	SW R30 R29 1236
218.	SW R29 R0 1300
219.	BEQ R0 R0 -120

The save/restore point creation process is also illustrated in Figure 8 as a flow chart.

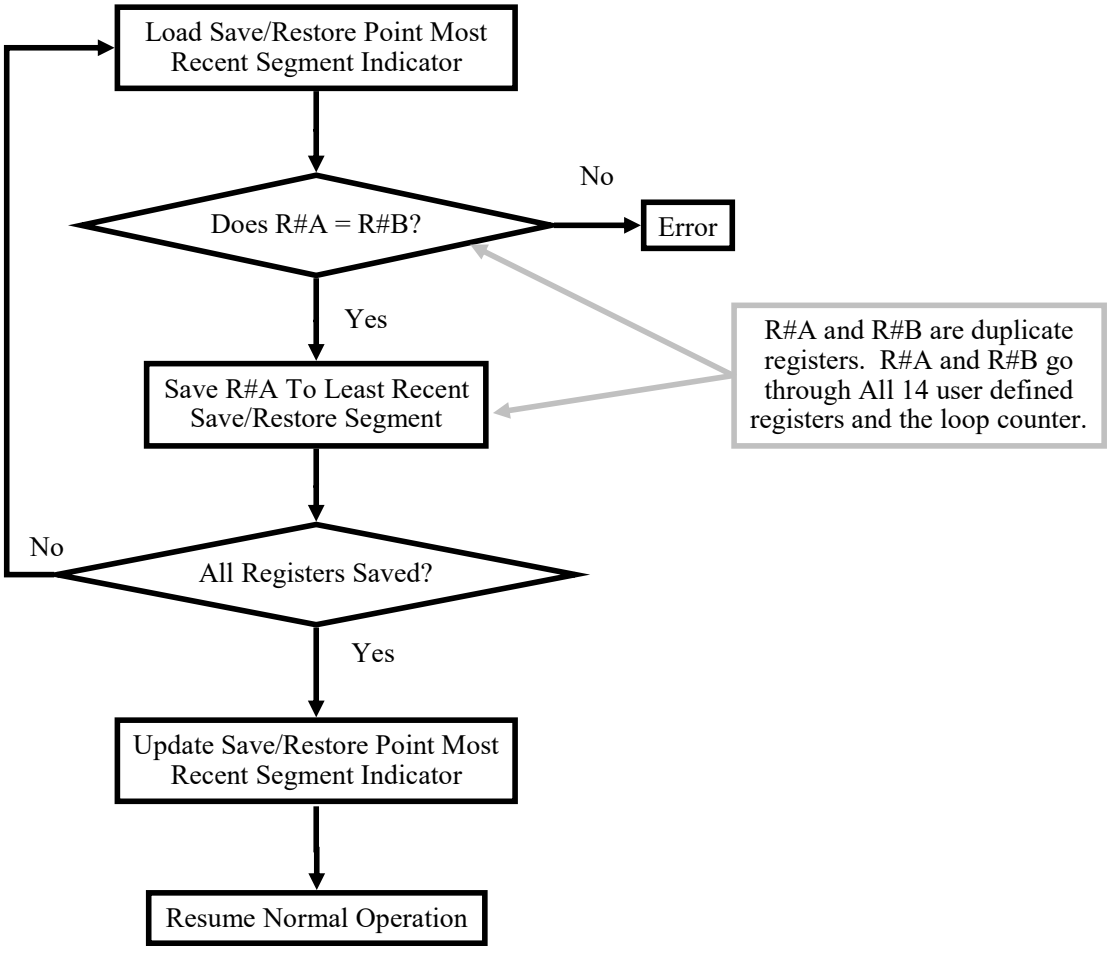

Figure 8. TSR MIPS Save/Restore Point Creation Flow Chart

Also note that the pointer to the most recently updated segment is repeatedly reloaded into R29. This is done because the only free register is R29 and there is no way in this TSR approach to duplicate R29 during the save/restore point creation process. The only way to protect R29 is to frequently refresh it, but that does not completely eliminate the possibility for error. Also note that the SW command always makes use of R29 as the base address and that the offset addresses always point to the first segment in the save restore point. Because the value of R29 is either 0 or 60 and the memory is byte addressable rather than word addressable (a word is four bytes), the value of 60 corresponds to 15 memory locations. This means that if R29 is 0 during the SW instruction, the store word instruction stores the register value to the first segment and if the value of R29 is 60, the SW instruction skips over the 15 memory locations in the first segment and stores the register value to the second segment. The final instruction is a branch to return code execution back to the end of the TSR MIPS program, just before the loop counter is decreased by 1. Note that this example of save/restore point creation code is associated with the TSR MIPS program in Table 4.

TSR MIPS also requires instructions to recover from errors called error recovery instructions. Error recovery instructions restore the internal state of TSR MIPS to the most recent save/restore point. The error recovery instructions examine the additional memory location in the save/restore point memory to determine which of the two segments is the most recently updated segment. It then issues load word instructions to load the variables and the loop counter from the segment into the user definable registers and duplicates. Once all variables and the loop counter have been stored in the Basic MIPS architecture, a branch command is issued to return code execution back to the end of the TSR program where the loop counter will be decreased before branching back to the beginning of the loop. Figure 9 is a flow chart

outlining the error recovery process.

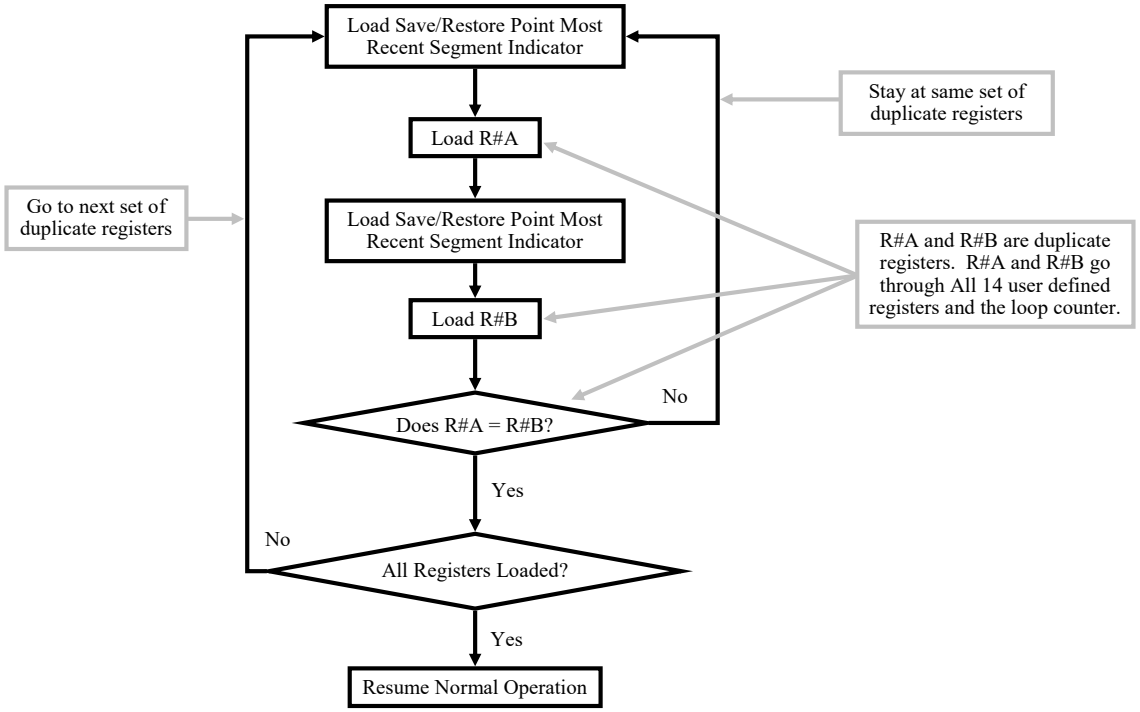

Figure 9. TSR MIPS Error Recovery Flow Chart

Table 6 shows a sample of the TSR error recovery instructions. A portion of the instructions have been omitted for brevity. The instructions start by loading the save/restore creation point additional memory location that indicates which segment is the most recent one. It then proceeds to load the value of R1 from the most recent save/restore point in memory to R1 in the Basic MIPS architecture. The value indicating the active segment is then reloaded from memory again before loading the value of R1 from the most recent save/restore point in memory to R15. A comparison is then performed to ensure R1 and R15 match, which they should if no error has occurred. If they do not match, the process is repeated by executing the branch command to go back to line 1. If no error occurs, the process is repeated for R2 and

R16. This continues for the remaining user registers and is then finally performed for the loop counter in lines 290 to 294. The branch instruction in line 295 causes code execution to return to the end of TSR instructions set, just before the loop counter is decreased by 1. Note that this example of error recovery code is associated with the TSR MIPS program in Table 4.

Table 6. TSR MIPS Error Recovery Code Example

Line Number	**Instruction**
220.	LW R29 R0 1300
221.	LW R1 R29 1180
222.	LW R29 R0 1300
223.	LW R15 R29 1180
224.	BNE R1 R15 -4
225.	LW R29 R0 1300
226.	LW R2 R29 1184
227.	LW R29 R0 1300
228.	LW R16 R29 1184
229	BNE R2 R16 -4
230 - 284	...
285.	LW R29 R0 1300
286.	LW R14 R29 1232
287.	LW R29 R0 1300
288.	LW R28 R29 1232
289.	BNE R14 R28 -4
290.	LW R29 R0 1300
291.	LW R30 R29 1236
292.	LW R29 R0 1300
293.	LW R31 R29 1236
294.	BNE R30 R31 -4
295.	BEQ R0 R0 -196

3.5 Adaptive-Hybrid Redundancy (AHR)

3.5.1 AHR Controller Finite State Machine

The development of Basic MIPS, TMR MIPS, TSR MIPS, and their associated programs lay the foundation for building a controller that can switch between TMR and TSR as needed. This controller, called the AHR Controller, is a Finite State Machine (FSM) that determines when to switch from TMR to TSR and vice versa. The AHR Controller also controls the orderly transition between the two operating modes. It is the backbone of AHR.

When the AHR Controller starts, it allows TMR MIPS to run. The AHR Controller continuously monitors the flow of information between the voter and memory without interrupting or delaying that flow and is able to determine whether TMR MIPS is operating normally or has encountered an error. The TMR Voter in AHR MIPS creates save/restore points after every 10,000 instructions, just as when it operates outside of AHR MIPS . TMR MIPS was modified slightly so that it provides an error signal to the AHR Controller that is 0 when no error is present and 1 when it has encountered an error.

The AHR Controller switches from TMR MIPS to TSR MIPS after TMR MIPS processes a predetermined number of instructions without an error. If an error occurs before the predetermined number of instructions are completed, a counter responsible for counting the number of error free instructions is reset to 0. The number of instructions TMR MIPS must complete without error before transitioning to TSR MIPS was arbitrarily set to be 15,000 initially. The transition process is complex and requires a review of the differences between TMR MIPS and TSR MIPS.

While TMR and TSR MIPS store the same data to permanent memory, there are marked differences between their temporary memory and their save/restore memory. Additionally, TMR MIPS can utilize 30 registers while TSR MIPS can only utilize 14;

this means that the internal states of TMR and TSR MIPS are incompatible. This also means there is no way to directly copy temporary memory, save/restore point memory, and internal state of TMR MIPS to TSR MIPS or vice versa. However, one unique feature of the programs for both TMR and TSR MIPS is the values in the registers in one loop iteration do not depend on the values in the registers of the previous loop iteration. The only exception is the loop counter that decreases by one on each loop iteration. The rule for all other registers is that their values from one loop iteration are overwritten in the next iteration. This unique feature points towards a solution. Only transition between TMR MIPS and TSR MIPS at the beginning or end of a loop. At this point, the values in all the registers can be ignored with the exception of the loop counter.

Note that this solution is unique to the chosen method of creating randomized programs for use in place of benchmarks such that results from one loop iteration does not impact the next loop iteration. In general, this is not the case for most real world programs. A more complicated and comprehensive approach would need to be implemented to ensure correct transition from TMR to TSR in real world applications, but this simplistic approach is sufficient for the purposes of this research. A real world approach might force any variables to be updated from one loop to the next to be stored in permanent memory on each loop iteration, and retrieved from permanent memory at the beginning of each loop iteration. If this were the case, the AHR Controller would still only be burdened with transitioning the loop counter from TMR to TSR and vice versa. The instructions in the loop would be responsible for loading and storing variables from one loop iteration to the next. This approach was not pursued due to time limitations and is considered out of scope of this research. Other, more sophisticated approaches that allow TMR to TSR and TSR to TMR transitions at any point in time may be possible, but are also out of scope of this

research.

When transitioning from TMR to TSR, the AHR Controller not only waits for the error free instruction counter to reach the predetermined number of instructions, but also waits for TMR MIPS to request access to the memory location corresponding to the start of the loop so that the transition will occur at the beginning of the loop. If an error is encountered after the predetermined number of instructions but before the start of the next loop, the AHR Controller will reset the instruction counter and the transition will not occur at the start of the next loop. Additionally, in order to successfully transition from TMR to TSR, a save/restore point must be created during the transition. The save/restore point ignores the values of all registers except the loop counter value, which is saved to the correct location in the TSR save/restore point.

The TMR to TSR transition begins with interception of all communications between the TMR Voter and the Memulator. The AHR Controller then instructs TMR MIPS to store the current loop count. The AHR Controller receives the loop count and adds 1 to the loop count. The AHR Controller then writes the loop count + 1 to both of the TSR save/restore point segments. The step of adding 1 to the loop count is important because TSR error recovery instructions return code execution to the TSR program at the end of the loop where the loop counter is decreased by one. If TSR were to encounter and recover from an error prior to creating a new save restore point and used the unmodified TMR loop counter value, the TSR instructions would immediately decrement the counter value and skip a loop that was never completed by TMR MIPS. The AHR Controller then sets the first save/restore segment to be the most recent segment by writing a 0 to the TSR save/restore additional memory location. The AHR Controller does not update the state of the registers because TSR MIPS will return to the start of the loop after error recovery and promptly overwrite

the values stored in the registers as a result of the error recovery process.

The AHR Controller then resets all three Basic MIPS processors, bypasses the TMR Voter, and begins sending signals to one of the three Basic MIPS processors while holding the other two processors' reset signals high and remaining inputs low to ensure they do not perform any processing actions. This effectively puts the two other processors in a low power mode. The AHR Controller prepares the one processor by instructing it to load the loop counter value to R31. This is the original loop counter value from TMR MIPS before adding 1 because TSR MIPS will start processing from the beginning of the TSR loop rather than at the end because the transition started at the beginning of the loop, not the end of the loop. After storing the loop counter in R31, a second command is sent to copy the value of R31 to R30 (ADDI R30 R31 0). The AHR Controller then issues a branch command to the Basic MIPS processor so that it will branch to the beginning of the TSR MIPS program loop. Once this is done, the AHR Controller then directly connects the Basic MIPS processor to the emulator. At this point, TSR MIPS is now operating and the AHR Controller begins monitoring communications between the Basic MIPS processor and the Memulator.

One particularly important communication between memory and TSR MIPS is the "DONE" signal. In AHR MIPS programs, the TMR instructions are first and start at memory address 0, followed by the TSR instructions. The "DONE" signal resets the Basic MIPS processor operating in TSR mode and causes its PC to reset to 0. If the AHR Controller takes no action, the Basic MIPS processor will begin processing the TMR instructions with no error mitigation. This is undesirable; therefore, the AHR Controller intercepts the first read command from the Basic MIPS processor to memory after the reset and issues a branch command to the Basic MIPS processor. The branch command tells the Basic MIPS processor to skip to the beginning of the TSR program so that TSR MIPS will continue to operate correctly after the reset.

The TSR to TMR transition occurs when TSR MIPS encounters two errors. If TSR MIPS encounters one error, it is permitted to carry out its error recovery operations. The AHR Controller has knowledge of the location of TSR MIPS save/restore code memory location and is able to determine that error recovery operations are in progress. The AHR Controller allows TSR MIPS to continue operations after recovering from the first error. If TSR MIPS successfully creates a save/restore point by executing its save/restore point creation instructions and returns to normal operation, the AHR Controller intentionally forgets that the error ever occurred. To rephrase, the AHR Controller sets a flag when TSR MIPS encounters an error and clears the flag when TSR MIPS creates a new save/restore point. If TSR MIPS encounters a second error before successfully creating a new save/restore point, that means that TSR MIPS would have to recover from the old save/restore point a second time. This indicates to the AHR Controller that the radiation environment is too severe to continue operations in TSR mode. It is likely that the error rate is high enough that TSR operations would continuously have to reset to an old save/restore point and would make little forward progress in running the program. In order for the program to continue running until completion, a more robust mitigation approach is needed, so the AHR Controller transitions from TSR to TMR. That way, a single error is more easily corrected without reverting to a previous save/restore point. The AHR Controller makes this transition and has TMR MIPS continue processing from the last TSR save/restore point.

The transition from TSR to TMR begins with resetting the Basic MIPS processor running in TSR mode. The next step is to determine the most recent save/restore point segment for TSR MIPS and load the loop counter stored in that segment. The loop count loaded from memory has one subtracted from it, then the result is stored to save/restore point memory locations 30 and 62; these are the memory locations

for the TMR MIPS save/restore point segments. Then the address of the start of the TMR program loop is saved to save/restore point memory locations 31 and 63; these memory locations store the PC value for the two save/restore point segments in TMR MIPS. The additional memory location for the TMR save/restore point is then updated to point to the first segment. The next step is to stop sending reset signals to the Basic MIPS processors and allow TMR MIPS to begin running. When TMR MIPS requests the first instruction from memory, the AHR Controller sends a load word command to load the loop counter, with one subtracted from it, into register R31, which TMR MIPS uses as the loop counter register. One is subtracted from the TSR save/restore point loop counter because TMR will begin at the start of the TMR MIPS instructions rather than at the end of the TMR MIPS instructions before decrementing the loop counter. After completing this step, a NOP command is issued to TMR MIPS so that its next instruction will be the start of the TMR program loop. A branch instruction could have been used instead to advance the PC, but branch instructions take longer to execute than NOP instructions and the branch would only need to advance the PC by one instruction. Finally, normal TMR operations are allowed to resume.

One final note is that it is assumed that the AHR Controller is hardened such that it is immune to errors. If the AHR Controller were subjected to SEUs and SETs, its internal state could be affected by radiation with adverse consequences to the behavior of AHR MIPS. A state error could place the AHR Controller in conflict with the TMR Voter and Basic MIPS processors in such a way that the AHR MIPS processor would freeze. A state error could also trigger a transition from TMR MIPS to TSR MIPS or vice versa when it is not the proper time for that transition to occur. A state error could also cause some steps of the transition process to be skipped. Another possibility is that a state error could jump from normal TMR or TSR operation to a

random transition state.

The Air Force Institute of Technology technical report on the AHR MIPS Architecture provides a more detailed look at the AHR Controller's FSM. This report can be made available upon request [35].

3.5.2 AHR MIPS Architecture

The previous section described the AHR Controller FSM that enables the transitions from TMR to TSR and vice versa. This section discusses how the AHR Controller integrates into the AHR MIPS Architecture. The AHR Controller provides inputs and select signals to a number of different multiplexers that control which signals flow between the Basic MIPS processors, the TMR Voter, and the Memulator. During TMR MIPS operations, the signals from the Basic MIPS processors are connected to the TMR Voter and the TMR Voter is connected to the Memulator through the multiplexers. During TSR MIPS operations, the signals from a single Basic MIPS processor are connected to the Memulator through the multiplexers while the other two Basic MIPS processors and TMR Voter are forced into a low dynamic power state by sending a high reset signal to them and holding all other input signals low. During the transition from TMR to TSR or TSR to TMR, the AHR Controller routes the flow of signals as needed to ensure an orderly transition as described in the previous section.

Figure 10 shows a diagram of how the AHR Controller is integrated into the AHR MIPS Architecture. All of the individual signals have been grouped together to provide a simplified overview of communications between the various components. For a more detailed view of how the AHR Controller exercises control over the various signals between the Basic MIPS processors, TMR Voter, and Memulator, please refer to Figure 69 in Appendix A. Figure 70 in Appendix A is a more detailed version of

Figure 10 which shows all of the signals between the Basic MIPS processors, TMR Voter, AHR Controller, and Memulator.

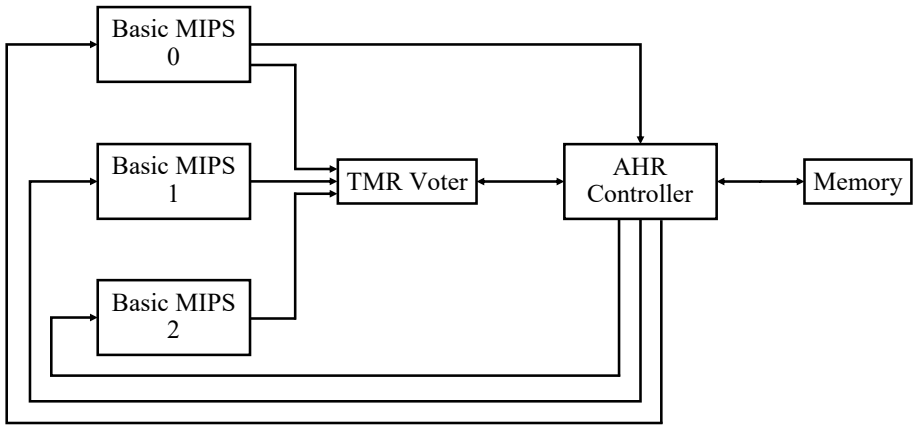

Figure 10. AHR MIPS Simplified Block Diagram

One final note is that it is assumed that the multiplexers used for signal routing are hardened such that they are immune to errors. If these multiplexers were vulnerable to SETs, an incorrect signal could be sent to the TMR Voter, Memulator, or any of the Basic MIPS processors resulting in undesirable behaviors. These behaviors could include faulty memory ready or reset signals sent to the TMR Voter; undesirable reads from or writes to memory; and incorrect memory ready or reset signals sent to the Basic MIPS processors. Any of these behaviors could cause AHR MIPS to freeze or experience errors that go unnoticed.

The Air Force Institute of Technology technical report on the AHR MIPS Architecture provides a more detailed look at the architecture. This report can be made available upon request [35].

3.5.3 AHR MIPS Programs

AHR MIPS requires instructions for both TMR MIPS and TSR MIPS. This is accomplished through simply placing the TMR MIPS program loop first, followed by

the TSR MIPS program loop, temporary memory, permanent memory, TSR save/restore point creation instructions, TSR error recovery instructions, and save/restore point memory. Because TMR and TSR differ in the amount of memory required for temporary memory and save/restore point memory, the larger of the two is used. TSR requires more temporary memory, so temporary memory space is allocated for TSR memory requirements. TMR requires more save/restore point memory, so save/restore point memory space is allocated for TMR save/restore point memory requirements. All store, load, and branch instructions are updated so that they point to the correct memory locations. Table 7 shows this graphically. The line numbers are used to illustrate the relative sizes of the various segments stored in memory. The line numbers come from an automatically generated AHR MIPS program. This set also happens to be equivalent to the Basic MIPS program example in Table 3 and the TSR MIPS program examples in Tables 4, 5, and 6. In fact, with the exception of the memory locations and line numbers, these are the exact same instructions. The Basic MIPS instructions from Table 3 are in lines 1 - 49 of Table 7. The TSR MIPS instructions from Table 4 are in lines 50 - 152 of Table 7. The TSR MIPS save/restore point creation instructions from Table 5 are lines 162 - 268 of Table 7. The TSR MIPS recovery instructions from Table 6 are in lines 269 - 344 of Table 7.

Table 7. AHR MIPS Program Structure

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

The AHR MIPS programs are automatically generated at the same time as the

TMR and TSR programs by combining the two sets and updating all memory location references in both TMR and TSR programs.

3.6 Summary

This chapter developed the AHR architecture beginning with the implementation of a simple MIPS-like architecture called Basic MIPS that was specifically developed for this research to allow for ease of integration into TMR, TSR, and AHR as well as to allow error injection. It further discussed the implementations of TMR and TSR and culminated in the development of AHR, which combines TMR and TSR in an architecture where the AHR Controller can switch between TMR and TSR depending upon the rate at which SEUs occur. This architecture is designed to answer the first research question, "Can multiple redundancy methods be incorporated into the redundancy design?" The next step will be to determine whether AHR functions as intended and answers the remaining research questions concerning mode switching, flexibility, and timing and energy tradespaces.

IV. AHR MIPS Performance Evaluation

4.1 Introduction

Now that the architecture for AHR has been developed, its function must be verified and its performance evaluated. This is necessary to answer the research questions, which are repeated here.

1. Can multiple redundancy methods be incorporated into the redundancy design?
2. Is it possible to allow flexibility in redundancy methods for the duration of a space vehicle's lifetime?
3. Is it possible to switch between these methods based on mission needs?
4. What are the timing and energy tradespaces available to a designer, mission planner, or operator?

The first step is functional verification of Basic MIPS, TMR MIPS, TSR MIPS, AHR MIPS, and their respective Memulators. Functional verification means that Basic MIPS, TMR MIPS, TSR MIPS, and AHR MIPS perform as designed in a simulated environment. These simulations were carried out in Mentor Graphics QuestaSim, which compiles the VHDL code and simulates it. Functional verification culminated in full system software simulations that verified each of the architectures could run a program to completion and successfully restart. Functional verification of Basic, TMR, TSR, and AHR MIPS is described in Section 4.2.

The next step is to perform timing analyses; however, there were limits to the software simulations. The simulations were limited in duration by the number of data points that RAM on the computer running the simulations could support. While the full system software simulations verified functionality, they could not directly provide

information on the total runtime of a program on each architecture. Fortunately, the software simulation does provide information on how long it takes to execute individual instructions, complete a loop in the program, create save/restore points, perform error recovery operations, and transition from TMR to TSR and TSR to TMR. This timing information, when combined with the known program, allows determination of program runtime. The method for performing the timing analysis is covered in Section 4.3.1. The results are discussed in Section 4.4.

In addition to timing analyses, energy analyses are also performed. Intel provides a spreadsheet tool to calculate the static and dynamic power used by a Cyclone V depending on the number of adaptive logic modules (ALMs) and registers a particular design uses. The spreadsheet tool is called the Intel PowerPlay Early Power Estimator for Cyclone IV and Cyclone V. When the power estimates are multiplied by the results of the timing analyses, estimates of the energy used to complete programs by the various architectures emerge. The method for performing the energy analysis is covered in Section 4.3.2. The results are discussed in Section 4.4.

While software simulations and analyses determine the runtimes and energy usage of an equivalent program across the architectures and allow comparisons between them, they do not provide any information on how they will perform when implemented in hardware. The next logical step is hardware-in-the-loop (HITL) simulations where the architectures and Memulators are implemented in hardware. The chosen hardware is the Intel Cyclone V FPGA implemented on a Terasic DE10-Standard board. These HITL simulations are considered simulations because they are operating in an error free environment and the FPGA is not necessarily the target technology, but rather used as an expedient for research purposes. The HITL simulations and results are discussed in Section 4.5.

4.2 Functional Verification

4.2.1 Basic MIPS Verification

Functional verifications began with the lowest levels of the Basic MIPS architecture and worked their way to the highest level modules. All functional verifications were conducted using Mentor Graphics QuestaSim software. The VHDL code for components and subcomponents of Basic MIPS along with testbench VHDL files to stimulate the inputs of the various components and subcomponents were compiled by and simulated in QuestaSim. Low level verifications were conducted on the NAND gate and D-flip-flop. These low level verifications were full factorial verifications; every possible input combination was attempted. Every NAND gate input combination (00, 01, 10, and 11) were attempted to verify the correct output (1, 1, 1, 0). The D-flip-flop was tested to ensure that every successive input was stored, whether the input order was 00, 01, 10, or 11. Full factorial verifications were conducted for all low level modules where the total number of input bits was fewer than nine (i.e. a 4-bit adder with two 4-bit inputs and one 1-bit input).

For higher level modules, full factorial tests were not conducted because there is insufficient time to conduct such tests. Instead, test patterns were loaded and results checked to ensure simulation results matched expected results for each test pattern (i.e. 32-bit adders, 32-bit shift operations, 32-bit AND, ALU, GPRs, and etc.). It was assumed that full factorial verifications of lower level modules when combined with test patterns for higher level modules was sufficient to verify correct operation of the higher level modules. For the Datapath module level, there are far too many inputs which must be synchronized to correctly verify its function; therefore, the Datapath functional verification was done in conjunction with overall Basic MIPS functional verification. Basic MIPS functional verification combined the Datapath and Controller to ensure they worked together correctly. If the Controller passes

verification and correctly articulates all of the Datapath's inputs, the Datapath's functional verification can be conducted in conjunction with Basic MIPS functional verification.

The Controller functional verification consisted of supplying the Controller with valid instructions when the Controller attempted to read an instruction from memory. Each of the 33 Basic MIPS instructions were verified. Branch instructions were verified to ensure that they worked correctly whether the branch should be taken or not taken. These inputs were much more manageable than the Datapath inputs. Additionally, erroneous instructions were also provided to ensure the Controller rejected such instructions and attempted to re-read the instructions from memory.

After Controller verification, the Datapath was connected to the Controller to form Basic MIPS. The same inputs that were supplied to the Controller for verification were supplied to Basic MIPS for verification. These verification tests uncovered some problems with the Datapath, the Controller, and the interface between the two. These problems were corrected, then Basic MIPS passed functional verification.

Functional verifications were also performed on the Memulator to ensure that it correctly performed read and write operations. These operations consisted of reading and writing patterns to a number of different memory addresses; however, these operations were not a full factorial test to test reading and writing to every possible 32-bit word to every possible 32-bit address. These operations were sufficient to provide Memulator verification.

Finally, the Memulator was combined with the Basic MIPS processor to ensure correct operation. The only inputs supplied by the testbench VHDL code were the master clock and master reset signals used by both Basic MIPS and the Memulator. Some errors were identified and corrected at this stage as well. After fixing these errors, the Basic MIPS processor was able to successfully read instructions from

memory and process them correctly. The processor was also able to successfully write to memory. The Memulator successfully reset itself and the processor upon program completion. To test program completion, the number of loops was reduced to three because the software simulation cannot handle doing 999 loops due to resource limitations of the computer running the software simulation.

4.2.2 TMR MIPS Functional Verification

TMR MIPS verification uncovered a couple of problems with Basic MIPS that were previously unidentified. These Basic MIPS problems were corrected and Basic MIPS was re-verified before resuming TMR MIPS verification.

The TMR Voter was verified as integrated into TMR MIPS rather than being verified independently due to its complexity. TMR MIPS was verified by allowing TMR MIPS to attempt to run Basic MIPS programs in a software simulation. These simulations enabled verification of correct TMR Voter state transitions during error free operations. Save/restore point creation was also verified. These simulations also ensured that TMR MIPS would reset correctly upon receiving a DONE signal from the Memulator at program completion. Some errors in TMR MIPS were uncovered and corrected as a result of the verification tests.

4.2.3 TSR MIPS Functional Verification

TSR MIPS functional verification revealed that some branch instructions used only by TSR MIPS programs were not being processed correctly by Basic MIPS. Basic MIPS branch processing was corrected, then Basic MIPS and TMR MIPS were re-verified before resuming TSR MIPS verification. After these corrections, TSR MIPS programs successfully executed on the Basic MIPS processor in software simulations. TSR MIPS save/restore point creation instructions were also verified.

The simulations also confirmed that TSR MIPS was correctly reset by the Memulator upon program completion. TSR MIPS error recovery instructions were verified once errors were injected into TSR MIPS.

4.2.4 AHR MIPS Functional Verification

AHR MIPS functional verification began with ensuring AHR MIPS allowed TMR MIPS to operate correctly since AHR MIPS begins in TMR mode after starting the processor. The next step was to verify the TMR to TSR transition in an error free software simulation. The TMR to TSR transition was also verified after some minor problems with the AHR Controller FSM were discovered and corrected. All registers and memory locations were correctly updated as the AHR Controller correctly transitioned between states. No additional problems were discovered in Basic MIPS, TMR MIPS, or TSR MIPS during these verification tests. The verification tests were also able to evaluate the TSR to TMR transition by using error injection.

4.3 Error Free Software Simulation

Error free software simulation is conducted to determine the time and energy it should take to run programs in Basic MIPS, TMR MIPS, TSR MIPS, and AHR MIPS when no errors are present. Software simulation cannot run an entire program (all the loops through a program) due to computer resource limitations, but software simulation can be used to determine the time to run individual instructions, complete a loop through the program, create save/restore points, and reset the processor upon program completion. The simulations can also be used to determine the additional time needed to transition from TMR to TSR MIPS in the AHR MIPS architecture. These times are utilized to calculate the time to complete any program on each architecture. The energy used to run each program is determined by multiplying

the dynamic power used by each architecture multiplied by the time used by the program run on each architecture. This dynamic power is determined using the Intel PowerPlay Early Power Estimator for Cyclone IV and Cyclone V. This tool takes the number of ALMs and registers used to implement the architecture and provides an estimate of the dynamic power used by the architecture.

The analyses are completed for a total of 1,000 randomly generated instructions sets. Each of the 1,000 programs has three variants. The Basic MIPS program is used by Basic MIPS and TMR MIPS. The TSR MIPS program is used by TSR MIPS. The AHR MIPS program is used by AHR MIPS. All three programs are equivalent in that they all compute the same values and results, and the same results are stored to permanent memory. The differences in the structures of the programs have been previously explained in Sections 3.4.2 and 3.5.3. An additional 40 equivalent TMR MIPS and TSR MIPS programs were generated for an early HITL simulation and analyses are also completed for these.

4.3.1 Time Simulation and Analysis

The first step in the time simulation and analysis was to determine the amount of time required for each type of Basic MIPS instruction to complete in Basic MIPS, TMR MIPS, TSR MIPS, and AHR MIPS. The next step was then to determine how long it took each program to run on each of these architectures.

The time to complete a Basic MIPS program is shown in Equation 1 where $T_{Basic\ MIPS}$ is the time to complete a Basic MIPS program, $T_{Basic\ init}$ is the time to set up the loop counter before looping through the program, n_{loops} is the number of loops, $T_{Basic\ loop}$ is the time to complete a single loop of the program and $T_{Basic\ conc}$ is the time to complete the program after completing the final program loop. The values for $T_{Basic\ init}$ and $T_{Basic\ conc}$ were measured from the software simulation. The

time to complete a single Basic MIPS program loop is given by Equation 2 where N_{Basic} is the number of instructions in the program loop and $t_{I_{Basic\ n}}$ is the amount of time to complete instruction number n. Measurements of $T_{Basic\ loop}$ were made to ensure that computing $T_{Basic\ loop}$ according to Equation 2 is a valid way of computing $T_{Basic\ loop}$.

$$T_{Basic\ MIPS} = T_{Basic\ init} + n_{loops} \cdot T_{Basic\ loop} + T_{Basic\ conc} \tag{1}$$

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

The time to complete a TMR MIPS program is shown in Equation 3 where $T_{TMR\ MIPS}$ is the time to complete a TMR MIPS program, $T_{TMR\ init}$ is the time to set up the loop counter before looping through the program, $T_{TMR\ loop}$ is the time to complete a single loop of the program, n_{SRP} is the number of save/restore points to create, $T_{TMR\ SRP}$ is the time to create a save/restore point, and $T_{TMR\ conc}$ is the time to complete the program after completing the final program loop. The values for $T_{TMR\ init}$, $T_{TMR\ SRP}$, and $T_{TMR\ conc}$ were measured from the software simulation. The time to complete a single TMR MIPS program loop is given by Equation 4 where N_{TMR} is the number of instructions in the program loop and $t_{I_{TMR\ n}}$ is the amount of time to complete instruction number n. Measurements of $T_{TMR\ loop}$ were made to ensure that computing $T_{TMR\ loop}$ according to Equation 4 is a valid way of computing $T_{TMR\ loop}$. The number of save/restore points to create is calculated as shown in Equation 5 where $n_{TMR\ init}$ is the number of instructions to initialize the loop counter, which is two for all TMR MIPS programs; $n_{TMR\ conc}$ is the number of instructions to complete the program after completing the final program loop, which is one instruction; and n_{save} is the number of instructions to complete before creating

a save/restore point, which is 10,000.

$$T_{TMR\ MIPS} = T_{TMR\ init} + n_{loops} \cdot T_{TMR\ loop} + (n_{SRP} - 1) \cdot T_{TMR\ SRP} + T_{TMR\ conc} \quad (3)$$

$$T_{TMR\ loop} = \sum_{n=1}^{N_{TMR}} t_{I_{TMR\ n}} \quad (4)$$

$$n_{SRP} = \left\lfloor \frac{n_{TMR\_init} + N_{TMR} \cdot n_{loops} + n_{TMR\_conc}}{n_{save}} \right\rfloor + 1 \quad (5)$$

The time to complete a TSR MIPS program is shown in Equation 6 where $T_{TSR\ MIPS}$ is the time to complete a TSR MIPS program, $T_{TSR\ init}$ is the time to set up the loop counter before looping through the program, n_{loops} is the number of loops, $T_{TSR\ loop}$ is the time to complete a single loop of the program, $T_{TSR\ SRP0}$ is the time to create a save restore point in the first save/restore point segment, $T_{TSR\ SRP1}$ is the time to create a save restore point in the second save/restore point segment, and $T_{TSR\ conc}$ is the time to complete the program after completing the final program loop. The values for $T_{TSR\ init}$, $T_{TSR\ SRP0}$, $T_{TSR\ SRP1}$ and $T_{TSR\ conc}$ were measured from the software simulation. The time to complete a single TSR MIPS program loop is given by Equation 7 where N_{TSR} is the number of instructions in the program loop and $t_{I_{TSR\ n}}$ is the amount of time to complete instruction number n. Measurements of $T_{TSR\ loop}$ were made to ensure that computing $T_{TSR\ loop}$ according to Equation 7 is a valid way of computing $T_{TSR\ loop}$. $T_{TSR\ skip}$ represents the time that would have been used by the instructions after the branch taken to create the save/restore point, but has been skipped because save/restore point creation always returns to the program instruction that decrements the loop counter. After creating three save/restore points, a total of three add immediate and three branch on equal instructions are

skipped. This time is calculated to be the sum of the these instructions and is not directly measured from simulation. Note that the first save/restore point that TSR MIPS creates is stored to the first segment of the save/restore point memory. Because TSR MIPS only creates three save/restore points when running a program, two save/restore points are stored to the second segment and one is stored to the first segment.

$$T_{TSR\ MIPS} = T_{TSR\ init} + n_{loops} \cdot T_{TSR\ loop} + T_{TSR\ SRP0} + 2 \cdot T_{TSR\ SRP1} + \cdots \\ T_{TSR\ conc} - T_{TSR\ skip} \tag{6}$$

$$T_{TSR\ loop} = \sum_{n=1}^{N_{TSR}} t_{I_{TSR\ n}} \tag{7}$$

The time to complete a AHR MIPS program is more complicated to compute. The timing largely depends upon the number of program loops completed in TMR and the number of loops to complete in TSR because TMR loops complete faster than TSR loops. Additionally, the number of save/restore points created in TMR mode depends on the number of instruction completed prior to the TMR to TSR transition while the number of save/restore points created in TSR mode depends on the loop count. The number of loops to complete before transitioning from TMR MIPS to TSR MIPS (P_{loops}) is determined according to Equation 8 where $n_{transition}$ is the number of instructions to complete without error in TMR MIPS before transitioning to TSR MIPS. The ceiling function is used because the transition must occur at the end of a loop after the transition point is reached. The number of instructions to initialize the TMR loop counter is subtracted from $n_{transition}$ because $n_{transition}$ includes $n_{TMR\ init}$ ($n_{TMR\ init}$ instructions must be completed prior to starting the loop, but $n_{transition}$ counts loop instructions and $n_{TMR\ init}$).

$$P_{loops} = \left\lceil \frac{n_{transition} - n_{TMR\ init}}{N_{TMR}} \right\rceil \tag{8}$$

After determining the loop number at which the TMR to TSR transition occurs, the next thing to determine is the number of TMR save/restore points to be created prior to the transition. This is calculated according to Equation 9 and is very similar to Equation 5.

$$n_{CSRP} = \left\lfloor \frac{n_{TMR\_init} + N_{TMR} \cdot P_{loops}}{n_{save}} \right\rfloor \tag{9}$$

The total time to complete a AHR MIPS program is then given by Equation 10 where $t_{AHR\ TMR}$ and $t_{AHR\ TSR}$ are the time spent in TMR and TSR mode respectively and $t_{TMR \rightarrow TSR}$ is the time required to transition from TMR to TSR. The time spent in TMR and TSR are separated to make the energy calculations simpler as will be seen in Equation 14.

$$
\begin{aligned}
&t_{AHR\ TMR} = t_{TMR\ init} + P_{loops} \cdot T_{TMR\ loop} + \cdots \\
&T_{TMR\ SRP} \cdot n_{CSRP} + t_{TMR \rightarrow TSR} \\
&if\ P_{loops}\ <\ 250 \\
&\quad t_{AHR\ TSR} = (n_{loops} - P_{loops}) \cdot t_{TSR\ loop} + T_{TSR\ SRP0} + \cdots \\
&\quad 2 \cdot T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip} \\
&elseif\ 250\ \leq\ P_{loops}\ <\ 500 \\
&\quad t_{AHR\ TSR} = (n_{loops} - P_{loops}) \cdot t_{TSR\ loop} + T_{TSR\ SRP0} + \cdots \\
&\quad T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip} \\
&elseif\ 500\ \leq\ P_{loops}\ <\ 750 \\
&\quad t_{AHR\ TSR} = (n_{loops} - P_{loops}) \cdot t_{TSR\ loop} + T_{TSR\ SRP1} + \cdots \\
&\quad T_{TSR\ conc} - \frac{2}{3} T_{TSR\ skip} \\
&elseif\ \ P_{loops}\ \geq\ 750 \\
&\quad t_{AHR\ TSR} = (n_{loops} - P_{loops}) \cdot t_{TSR\ loop} + T_{TSR\ conc} \\
&end \\
&T_{AHR\ MIPS} = t_{AHR\ TMR} + t_{AHR\ TSR}
\end{aligned}
\tag{10}
$$

The run times for Basic MIPS are expected to be the fastest, followed by TMR MIPS, then AHR MIPS, and finally TSR MIPS. TMR MIPS is expected to take longer to run than Basic MIPS due to the overhead of running all memory read and write operations through the TMR Voter as well as the need to create save/restore points in memory. TSR MIPS should require the longest run time because all instructions are duplicated and there is the additional overhead of the creating save/restore points in memory. AHR MIPS is expected to run faster than TSR MIPS because AHR MIPS starts in TMR MIPS mode, but slower than TMR MIPS because AHR MIPS transitions to TSR MIPS after completing a predetermined number of instructions without error.

4.3.2 Energy Analysis

Equation 11 shows how Basic MIPS energy usage is computed for each program where $P_{Basic\ MIPS}$ is the dynamic power used by Basic MIPS as calculated using the PowerPlay tool.

$$E_{Basic\ MIPS} = P_{Basic\ MIPS} \cdot T_{Basic\ MIPS} \tag{11}$$

Equation 12 shows how TMR MIPS energy usage is computed for each program where $P_{TMR\ MIPS}$ is the dynamic power used by TMR MIPS as calculated using the PowerPlay tool.

$$E_{TMR\ MIPS} = P_{TMR\ MIPS} \cdot T_{TMR\ MIPS} \tag{12}$$

Equation 13 shows how TSR MIPS energy usage is computed for each program. Note that $P_{Basic\ MIPS}$ is used instead of TSR specific power because TSR MIPS uses a single Basic MIPS processor, but runs a different program.

$$E_{TSR\ MIPS} = P_{Basic\ MIPS} \cdot T_{TSR\ MIPS} \tag{13}$$

Equation 14 shows how AHR MIPS energy usage is computed for each program where $P_{CTMR\ MIPS}$ is the dynamic power used by AHR MIPS when operating in TMR as calculated using the PowerPlay tool and $P_{CTSR\ MIPS}$ is the dynamic power used by AHR MIPS when operating in TSR as calculated using the PowerPlay tool. Unfortunately, PowerPlay simply provides the overall power usage $P_{AHR\ MIPS}$ because it does not understand that AHR MIPS effectively shuts down two processors and a voter when operating in TSR mode. When operating in TMR mode, the full $P_{AHR\ MIPS}$ equals the amount of power used by TMR $P_{CTMR\ MIPS}$. The correct value for $P_{CTSR\ MIPS}$ is given by Equation 15 where the difference between $P_{CTMR\ MIPS}$

and $P_{TMR\ MIPS}$ should be the power used by the AHR Controller. This difference is added to the power used by a single Basic MIPS processor to determine how much power AHR MIPS uses in TSR mode. It will be shown in Section 4.4 that there is no difference in the PowerPlay results for $P_{AHR\ MIPS}$ and $P_{TMR\ MIPS}$, which means that $P_{CTSR\ MIPS} = P_{TSR\ MIPS}$.

$$E_{AHR\ MIPS} = P_{CTMR\ MIPS} \cdot t_{AHR\ TMR} + P_{CTSR\ MIPS} \cdot t_{AHR\ TSR} \tag{14}$$

$$P_{CTSR\ MIPS} = P_{CTMR\ MIPS} - P_{TMR\ MIPS} + P_{Basic\ MIPS} \tag{15}$$

The energy used by Basic MIPS is expected to be the lowest, followed by TSR MIPS, then AHR MIPS, and finally TMR MIPS. TSR MIPS is expected to take at least twice as much energy to run as Basic MIPS due to the instruction duplication and additional instructions to create save/restore points. TSR MIPS dynamic power is identical to Basic MIPS, so the main cause of the energy increase is the increased runtime. TMR MIPS is expected to use at least three times as much power as Basic MIPS because it uses three processors and a voter. Additionally, the added time caused by using the TMR Voter and creating save/restore points will add to the energy use. AHR MIPS is expected to use more energy than TSR MIPS and less energy than TMR MIPS because it divides its operating time between the two.

4.4 Error Free Software Simulation Results

This section begins by showing the timing results for individual instructions in Basic MIPS, TMR MIPS, TSR MIPS, and AHR MIPS. It also provides the timing results for several key performance parameters needed to compute program runtimes.

Next, the results of the Intel PowerPlay Early Power Estimator are presented.

This is followed by a timing and energy analysis for the 40 TMR MIPS and TSR MIPS programs used in the First Attempt at Error Free HITL Simulation described in Section 4.5.2. It then continues with the timing and energy analysis of an additional 1,000 programs for Basic MIPS, TMR MIPS, TSR MIPS, and AHR MIPS.

The timing results for individual instructions is provided in Table 8 and all times are given in nanoseconds. The instruction names are in column one. The timing for Basic MIPS, TSR MIPS, and AHR MIPS operating in TSR mode are provided in column two. The timing for TMR MIPS and AHR MIPS operating in TMR mode is in column three. From this table, it is apparent that the inclusion of the TMR Voter adds 100ns (five clock cycles) for each memory interaction. Most instructions have only one interaction, but the store word and load word instructions have two memory interactions, which is why these two instructions take 200ns (ten clock cycles) longer in TMR than in TSR. This is in agreement with how the TMR Voter was designed [38]. TSR MIPS has no hardware voter and suffers no hardware delay. TSR MIPS instead has branch instructions prior to store word instructions to compare duplicate registers and check for errors. These instructions add delay to the overall program, but not the execution time of individual instructions when compared to Basic MIPS.

Table 8. Software Simulation Individual Instruction Timing Results in Nanoseconds

Instruction	**Basic MIPS TSR MIPS AHR MIPS TSR**	**TMR MIPS AHR MIPS TMR**
SLL	160	260
NOP	160	260
SRL	160	260
SRA	160	260
SLLV	160	260
SRLV	160	260
SRAV	160	260

Table 8 – *Continued on next page*

Table 8 – *Continued from previous page*

Instruction	**Basic MIPS** / **TSR MIPS** / **AHR MIPS TSR**	**TMR MIPS** / **AHR MIPS TMR**
ADD	160	260
ADDU	160	260
SUB	160	260
SUBU	160	260
AND	160	260
OR	160	260
XOR	160	260
NOR	160	260
SLT	160	260
SLTU	160	260
BGEZ	180	280
BLTZ	180	280
BEQ	180	280
BNE	180	280
BLEZ	180	280
BGTZ	180	280
ADDI	160	280
ADDIU	160	260
SLTI	160	260
SLTIU	160	260
ANDI	160	260
ORI	160	260
XORI	160	260
LUI	160	260
LW	280	480
SW	280	480

The aforementioned key performance parameters needed to compute program runtimes are provided in Table 9. Also note that the TSR to TMR transition time ($TSR \rightarrow TMR$) has been added to this table. While this value is not used in the evaluation of AHR MIPS because the expected error rate is sufficiently low that a TSR to TMR transition should never occur, it was still determined using software simulation with error injection during the process of verifying that the TSR to TMR

could be performed correctly.

[H]

Table 9. Software Simulation Key Timing Parameter Results in Nanoseconds

Key Timing Parameter	Time (ns)
$T_{Basic\ init}$	320
$T_{Basic\ conc}$	180
$T_{TMR\ init}$	520
$T_{TMR\ conc}$	280
$T_{TMR\ SRP}$	11,840
$T_{TMR\ ttdA}$	380
$T_{TMR\ recA}$	13,920
$T_{TMR\ retA}$	20
$T_{TMR\ repA}$	480
$T_{TMR\ ttdB}$	320
$T_{TMR\ recB}$	11,600
$T_{TMR\ SRP\ Err}$	11,600
$T_{TSR\ init}$	480
$T_{TSR\ conc}$	180
$T_{TSR\ SRP0}$	16,660
$T_{TSR\ SRP1}$	19,360
$T_{TSR\ skip}$	1,020
$T_{TSR\ Rec}$	19,680
$T_{TSR\ SRP0\ Err}$	15,920
$T_{TSR\ SRP1\ Err}$	18,620
$T_{TMR \rightarrow TSR}$	1,620
$T_{TSR \rightarrow TMR}$	1,980

The Intel PowerPlay Early Power Estimator was provided with the Adaptive Logic Module (ALM) and register counts for Basic MIPS, TMR MIPS, TSR MIPS, and AHR MIPS. The register ALM and register counts are shown along with the dynamic power estimate in Table 10. Note that the power for TSR MIPS is identical to the power for Basic MIPS because the architectures are identical. TSR MIPS is a Basic MIPS processor that runs TSR instructions using Error Detection by Duplicated Instructions as discussed in Section 2.3.2.2 [71]. Additionally, Equation 15 is used

to compute $P_{CTSR\ MIPS}$ in Equation 16 which is identical to $P_{Basic\ MIPS}$ because $P_{CTMR\ MIPS} = P_{TMR\ MIPS}$ according to the PowerPlay tool. Also note that in this equation, $P_{CTMR\ MIPS} = P_{AHR\ MIPS}$ as discussed previously in Section 4.3.

Table 10. PowerPlay Results

Architecture	ALMs	Registers	Dynamic Power (mW)	Variable
Basic MIPS	1,664	1,076	9	$P_{Basic\ MIPS}$
TMR MIPS	5,455	5,391	29	$P_{TMR\ MIPS}$
AHR MIPS	5,748	5,200	29	$P_{AHR\ MIPS}$

$$\begin{aligned} P_{CTSR\_MIPS} &= P_{CTMR\ MIPS} - P_{TMR\ MIPS} + P_{Basic\ MIPS} \\ P_{CTSR\_MIPS} &= P_{Basic\ MIPS} = 9mW \end{aligned} \tag{16}$$

The next step is to compute the error free software simulation times for the 40 TMR MIPS and 40 TSR MIPS programs used in the first HITL simulation attempt according to Equations 3 and 6. The energy used by these programs is computed using Equations 12 and 13. The results are presented in Figure 11 where the energy used to complete each program is plotted against the time to complete each program. Also shown is the average time and energy to complete all programs for each architecture. Both the TMR MIPS and TSR MIPS programs fall on lines. This makes sense because the energy to complete a program is the dynamic power for the architecture upon which the program is run, TMR MIPS and TSR MIPS, multiplied by the time needed to complete each program; therefore, the slope of these lines equal the dynamic power for TMR and TSR MIPS respectively.

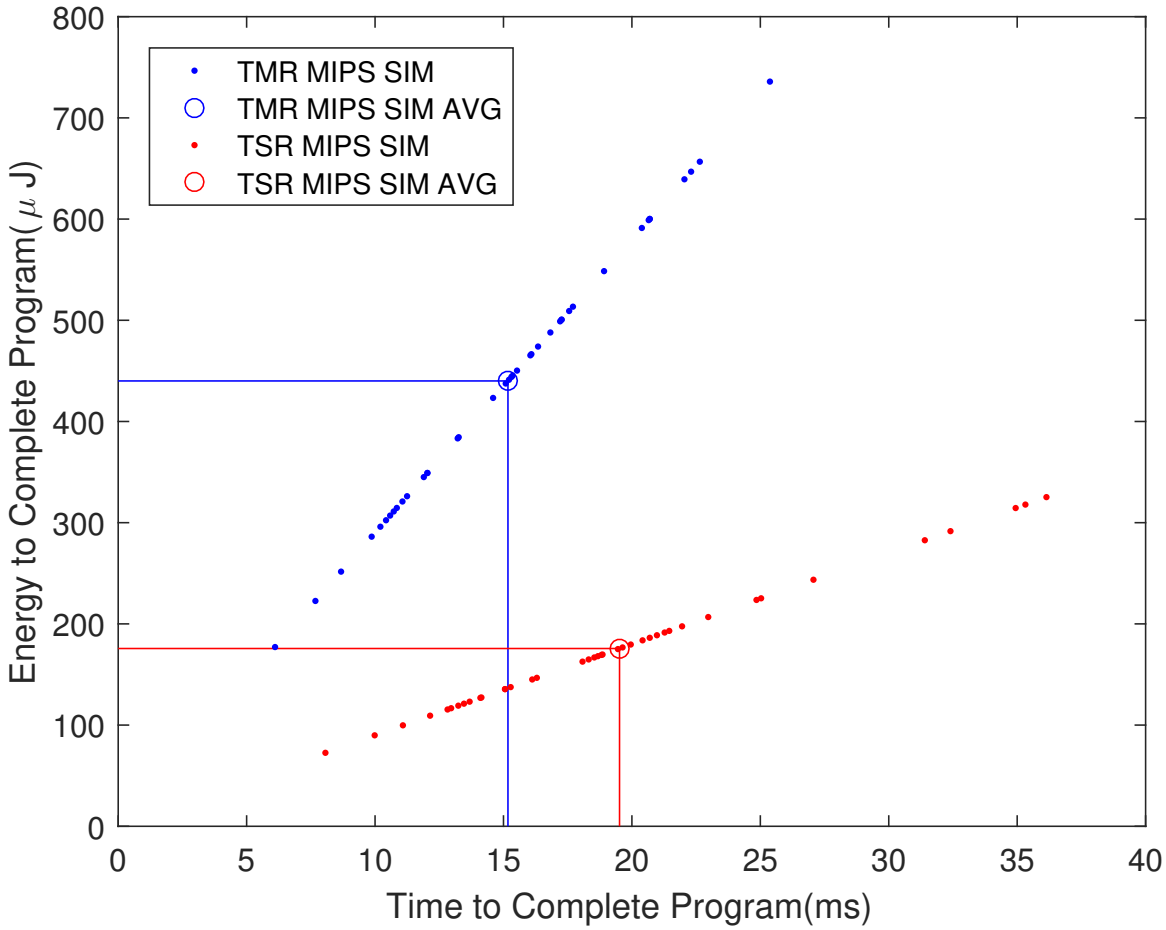

Figure 11. First HITL Attempt Software Simulation Energy vs. Time to Complete

After the first HITL attempt, an additional 1,000 programs were generated for Basic MIPS, TMR MIPS, TSR MIPS, and AHR MIPS. The TMR to TSR transition point for AHR MIPS was set at 15,000 instructions. The timing and energy computations for the TMR MIPS and TSR MIPS programs were performed the same way as for the first HITL attempt programs. The Basic MIPS and AHR MIPS timing were computed according to Equations 1, 2, 8, 9, and 10. The Basic MIPS and AHR MIPS energy were computed according to Equations 11 and 14. The results of these computations are presented in Figure 12 where the energy used to complete each program is plotted against the time to complete each program. Also shown is the average time to complete all programs for each architecture.

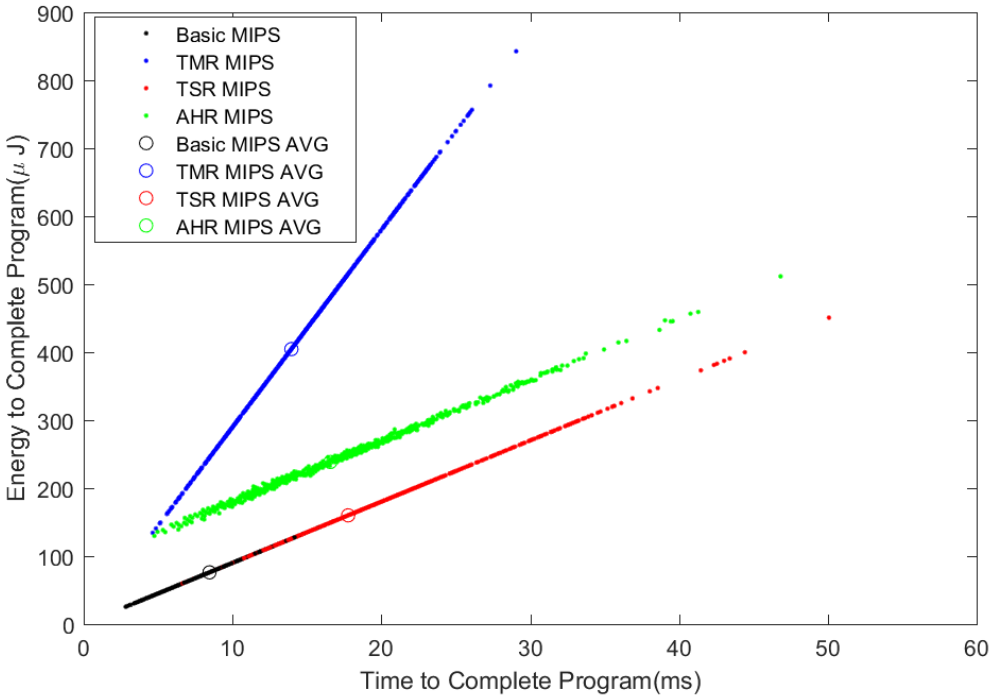

Figure 12. Error Free Software Simulation Energy vs. Time to Complete

These simulation results meet the expectation that Basic MIPS programs will complete faster and use less energy than programs running on the other architectures. TMR MIPS completes faster than AHR MIPS, and AHR MIPS completes faster than TSR MIPS as expected. TMR MIPS uses more energy than AHR MIPS, and AHR MIPS uses more energy than TSR MIPS as expected. These differences are further quantified as percent differences when compared to Basic MIPS. The percent differences were calculated for individual programs, then those percent differences were averaged to determine an average percent difference. The formulas for these calculations are shown in Equations 17, 18, 19, 20, 21, and 22.

$$PD_{Time\ TMR\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{TMR\ MIPS}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{17}$$
$$PD_{Time\ TMR\ v\ Basic} = 65.16\%$$

$$PD_{Time\ TSR\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{TSR\ MIPS}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{18}$$
$$PD_{Time\ TSR\ v\ Basic} = 109.40\%$$

$$PD_{Time\ AHR\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{AHR\ MIPS}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{19}$$
$$PD_{Time\ AHR\ v\ Basic} = 93.70\%$$

$$PD_{Energy\ TMR\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{TMR\ MIPS}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{20}$$
$$PD_{Energy\ TMR\ v\ Basic} = 432.19\%$$

$$PD_{Energy\ TSR\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{TSR\ MIPS}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{21}$$
$$PD_{Energy\ TSR\ v\ Basic} = 109.40\%$$

$$PD_{Energy\ AHR\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{AHR\ MIPS}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{22}$$
$$PD_{Energy\ AHR\ v\ Basic} = 224.03\%$$

The 65% timing difference between TMR MIPS and Basic MIPS is consistent with the expectation that TMR MIPS would take longer to run because of the delay introduced by the TMR Voter. The 109% timing difference between TSR MIPS and Basic MIPS is consistent with the expectation that TSR MIPS would take at least twice as long as Basic MIPS. The 94% timing difference between AHR MIPS and Basic MIPS is consistent with the expectation that AHR MIPS programs would take more time than TMR MIPS and less time than TSR MIPS to complete because it divides its time between TMR and TSR modes.

The 432% energy difference between TMR MIPS and Basic MIPS is consistent with the expectation that TMR MIPS would use at least three times the energy of Basic MIPS because it uses three times as many processors and the TMR Voter adds delay. The 109% energy difference between TSR MIPS and Basic MIPS is consistent with the expectation that TSR MIPS would use at least twice as much energy as Basic MIPS because it takes at least twice as much time as Basic MIPS. This is reflected in the fact that the timing and energy percent differences between TSR MIPS and Basic MIPS are identical. Finally, the 224% energy difference between AHR MIPS and Basic MIPS is consistent with the expectation that AHR MIPS would use more energy than TSR MIPS and less energy than TMR MIPS because it divides its time between TMR and TSR modes.

One thing that may not be evident from the analysis of Figure 12 and the percent differences is that AHR MIPS can be adjusted. This adjustment can be made by changing the number of instructions AHR MIPS must complete without error in TMR mode before transitioning to TSR mode. If this transition point is made sufficiently large, no transition to TSR would ever occur for most programs and the AHR MIPS results would match the TMR MIPS results. If this transition point is set to 0, the transition to TSR would occur at the beginning of every program and AHR MIPS

results would match the TSR MIPS results. This effect can be observed in Figure 12 by more closely examining the two endpoints of the AHR MIPS results on the left and right. On the left are shorter programs that spend most of their time in TMR mode and very little time in TSR mode. The transition occurs towards the very end of these programs. The results of these programs more closely match the results of TMR MIPS than TSR MIPS. On the right are longer programs that spend much more time in TSR mode than TMR mode. The transition occurs towards the beginning of these programs. The results of these programs more closely match the results of TSR MIPS than TMR MIPS. This highlights the versatility of AHR in allowing a space vehicle designer, mission planner, or operator to adjust the transition point as needed to perform more of the program in TMR or more in TSR.

It is also possible to examine how changing the TMR to TSR transition point affects the average AHR MIPS program completion time and energy usage. Figure 13 illustrates what happens to AHR MIPS average completion time and energy usage if the TMR to TSR transition point is varied from 11,000 instructions to 80,000 instructions in increments of 1,000. When the TMR to TSR transition point occurs at 11,000 instructions, the average AHR MIPS completion time is around 17ms and the average energy to complete is about 200μJ. As the transition point increases, the AHR MIPS average completion time moves up and to the left, and approaches the TMR MIPS average completion time of approximately 14ms and completion energy of approximately 400μJ. This further highlights the versatility of AHR in allowing a space vehicle designer, mission planner, or operator to adjust the transition point as needed to perform more of the program in TMR or more in TSR to achieve faster processing speeds or more energy efficiency.

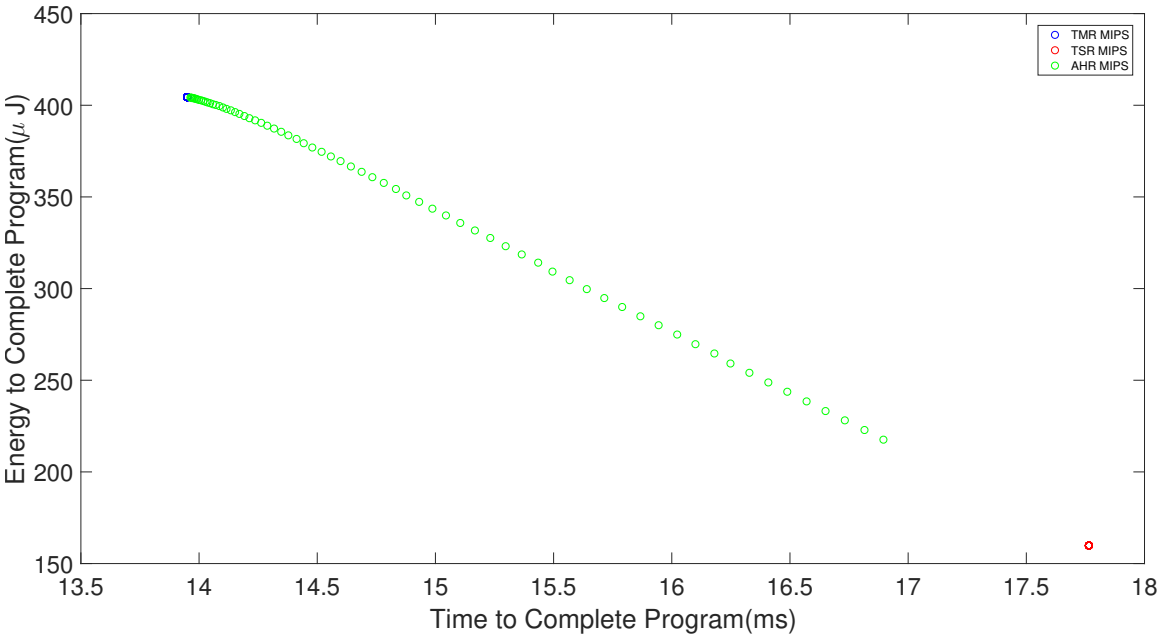

Figure 13. AHR MIPS TMR to TSR Transition Varying from 11,000 to 80,000 Instructions - Energy vs. Time to Complete

4.5 Error Free HITL Simulation

4.5.1 First Attempt Methodology

The first error free HITL simulations placed a processor and Memulator on the same Intel Cyclone V FPGA. The early HITL simulations were performed on a Terasic SoCKit (10-31212180-C0) development board using an Altera Cyclone V chip (5CSXFC6D6F31C8NES) rather than on the Terasic DE10-Standard board using an Altera Cyclone V (5CSXFC6D6F31C6N), which was procured after the early HITL simulations were completed. These early HITL simulations evaluated the timing and energy consumption TMR MIPS and TSR MIPS for 40 equivalent programs (40 TMR MIPS and 40 equivalent TSR MIPS programs).

Measurements of the energy usage were made using a Keysight DSOS054A Infiniium S-Series Digital Storage Oscilloscope. To do so, a Keysight N2873A Voltage Probe was used to measure the voltage at the power input to the SoCKit and the

Keysight N2821A Current probe fitted with the N2824A probe head capable of 50μA resolution in series with the power supply to the SoCKit as shown in Figure 14, where "V" indicates the voltage probe and "A" indicates the current probe. The physical connections were made using a breadboard and are shown in Figure 15. This allowed instantaneous power measurements. A second Keysight N2873A Voltage Probe is used to measure the DONE signal to provide triggering for the oscilloscope as seen in Figure 16. The DONE signal is provided as an output via the Terasic HSTC to GPIO Daughter Board since there is no easy way to produce the DONE signal on a pin on the SoCKit in such a way that the voltage probe can measure it. The DONE signal is assigned to PIN_D2 of the Altera Cyclone V chip which corresponds to SoCKit signal HSMC_TX_p9. HSMC_TX_p9 is connected to pin 107 of the SoCKit HSMC port [99]. This connects to pin 53 on the HSTC to GPIO Daughter Card because of the manner in which the male and female connectors pins are assigned numbers. Pin 53 corresponds to signal HSTC_TX_p7 which appears as pin 40 on the connector named "J2" on the daughter card [100].

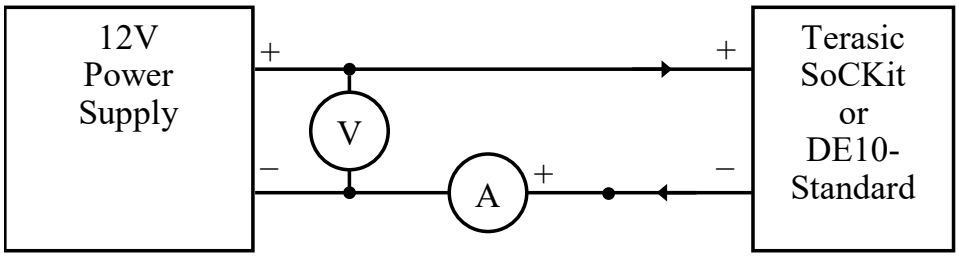

Figure 14. HITL Simulation Current and Voltage Measurement Setup

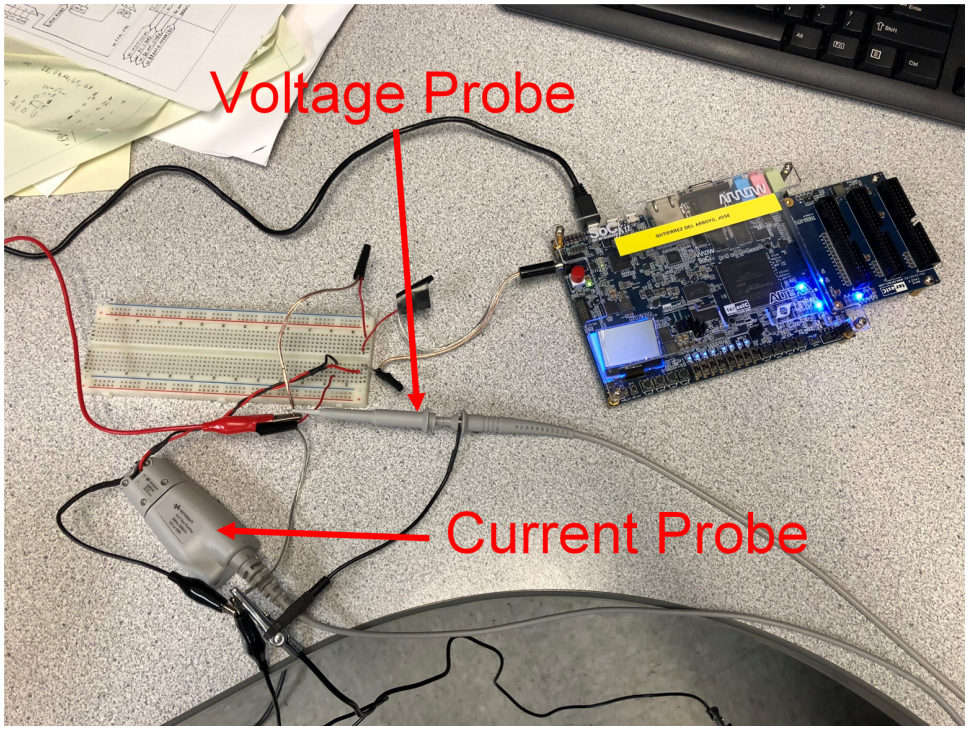

Figure 15. Voltage and Current Probe Connections

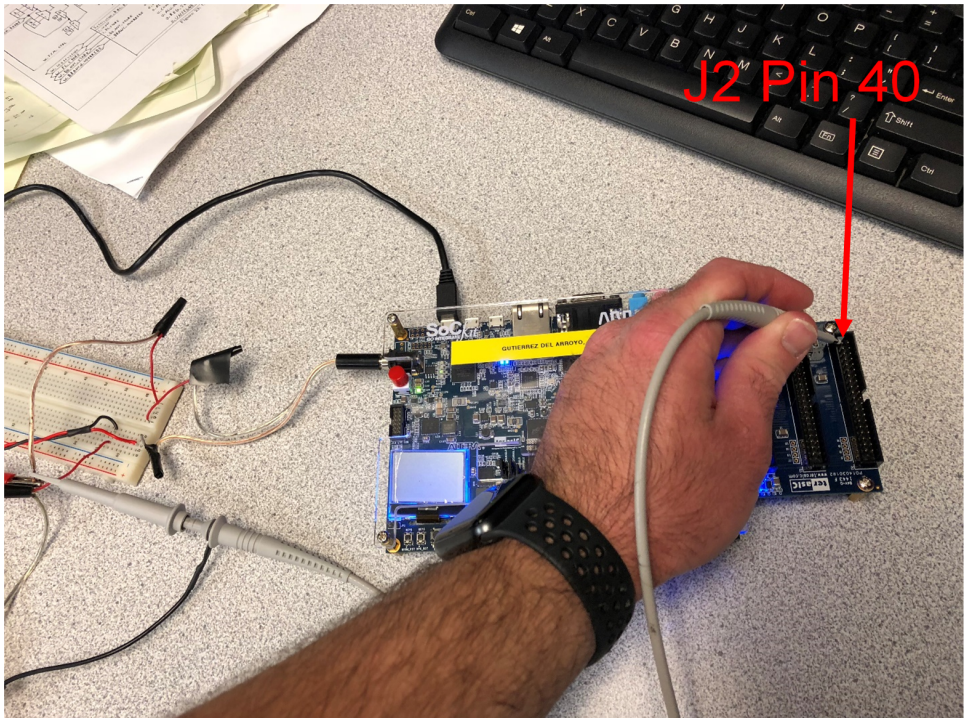

Figure 16. DONE Signal Oscilloscope Connection

There were no calibration steps outlined in the oscilloscope user manual [97]. The calibration guide for setting up the N2873A voltage probe [96] was strictly followed. An attempt was made to follow the calibration procedures for the current probe as outlined in the N2821A user manual [98]; however, one of the steps called for following the prompts on the oscilloscope. The oscilloscope prompts called for an E2655 PV/Deskew Fixture which was unavailable at the time. However, a certificate of calibration was received with the probe when it was purchased. It is possible that calibration errors could have been introduced into the probe during shipment, but there is no way of detecting or correcting those errors at this time. The user manual's guidance on allowing the current probe to warm-up for at least 15 minutes prior to taking any measurements was followed. The grounding procedures laid out in the DSOS054A, N2873A, and N2821A user manuals [96, 97, 98] were also strictly followed. Additionally, the FPGA was allowed to run each architecture type and program for at least five minutes prior to taking a measurement to alleviate any temperature changes on the FPGA due to running different hardware for each program. An Extech Instruments 3822000 DC Power Supply was substituted in place of the power supply provided by Terasic with the SoCKit; an EDAC Power Electric model EA10402E-120 DC supply with a marketed ripple/noise level of 250mV [18] and an observed ripple/noise of 235mV. This decision was made after some initial testing with the oscilloscope, voltage probe, and current probe revealed that the Terasic power supply was too noisy to make useful voltage and current measurements. The current also had an observed ripple/noise of 67mA. The frequency content of the voltage and current ripple was much lower than the 50MHz FPGA clock speed and it appeared to be consistent with the type of ripple this author has previously observed in some switching power supplies.

In addition to the hardware testing for both TMR MIPS and TSR MIPS, a very

simple VHDL program was also created that would connect one of the FPGA push buttons to one of the LEDs. The push button is active low so that it is 0 when pressed. The push button was connected to a single inverter so that the LED is on when the push button is pressed and off when it is not pressed. The purpose of this program is to make the absolute minimal use of the FPGA so that current and voltage measurements can be taken to establish a baseline for FPGA energy consumption. By doing this, the baseline energy can be subtracted from the measured energy for TMR and TSR MIPS. This eliminates any energy used by the SoCKit and FPGA that is always used regardless of the hardware implemented on the FPGA.

The data collection methodology consisted of running the simple program on the FPGA for about five minutes prior to collecting a baseline reading for a TMR MIPS program. While the baseline reading was saving, the FPGA was reprogrammed with a TMR MIPS processor and Memulator. The baseline save took at least five minutes, after which time two data sets were recorded on a TMR MIPS program. Each data set showed the DONE signal pulsing six times, which means the program completed five times. Five program completions were recorded for every data set by setting the timescale on the oscilloscope to record five program completions based on the timing results of the simulations and analyses. Then, another baseline reading was taken before recording two data sets for the equivalent TSR MIPS program. This second baseline serves as a baseline for the TSR MIPS program. This procedure recorded a total of ten program runs for each of the 40 programs for TMR MIPS and TSR MIPS. The total amount of data collected were 80 baselines, 400 TMR MIPS program runs, and 400 TSR MIPS program runs. For each of these, voltage and current were collected as well as the time to complete each program.

It was important to follow this procedure to minimize temperature variations between TMR MIPS and TSR MIPS program data collects. Early experimentation

showed a significant change in baseline FPGA energy usage over time as the temperature varied in the room in which testing was performed. The room had very poor temperature controls.

After collecting this data, the time to complete each program run was determined. The instantaneous power at every point in time for every program run was calculated according to Equation 23, then integrated over time using trapezoidal integration to determine the amount of energy used as shown in Equation 24. In these equations, the subscript TXR_N_T indicates whether the program was a TMR MIPS or TSR MIPS program depending on whether "X" is "M" or "S" respectively, the "N" is the instruction set number (i.e. 1-40), and "T" is the program run number (i.e. 1-10). The variables $\vec{P}$, $\vec{V}$, and $\vec{I}$ are vectors containing the instantaneous power, voltage, and current at every point in time for a particular program run of a TMR or TSR MIPS program number. The variable $\vec{t}$ is the time vector for a particular program run of a TMR or TSR MIPS program number. The variable E represents energy. A variable with a bar over it such as $\bar{E}$ or $\bar{t}$ denotes an average value. The "$\cdot$" operator in Equation 23 denotes an elementwise multiplication. The "M" in Equation 24 refers to the number of data points collected during the program run, t refers to the time at the specified index, and m is the indexing variable. The time and energy to complete the 10 program runs for the same program were averaged to determine the average time and average energy to complete each program as shown in Equations 25 and 26 respectively. Note that energy calculated here was the energy used by the entire Terasic SoCKit board, and not just by the particular program running on the FPGA. As a result, the baseline Terasic SoCKit board energy usage needs to be determined, and subtracted from the energy used by the Terasic SoCKit board running TMR or TSR MIPS on the FPGA. That is the rationale behind including "SOC" in the subscripts for power and energy as these values include baseline power and energy in

addition to that used by the TMR or TSR MIPS running on the FPGA.

$$\vec{P}_{SOC\_TXR\_N\_T} = \vec{V}_{TXR\_N\_T} \cdot \vec{I}_{TXR\_N\_T} \tag{23}$$

$$E_{SOC\_TXR\_N\_T} = \sum_{m=1}^{M-1} \left[\left(\frac{\vec{P}_{SOC\_TXR\_N\_T}(m) + \vec{P}_{SOC\_TXR\_N\_T}(m+1)}{2} \right) \times \left(\vec{t}_{TXR\_N\_T}(m+1) - \vec{t}_{TXR\_N\_T}(m) \right) \right] \tag{24}$$

$$\bar{t}_{TXR\_N} = \frac{\sum_{N=1}^{10} \left(\vec{t}_{TXR\_N\_T}(M) - \vec{t}_{TXR\_N\_T}(1) \right)}{10} \tag{25}$$

$$\bar{E}_{SOC\_TXR\_N} = \frac{\sum_{N=1}^{10} E_{SOC\_TXR\_N\_T}}{10} \tag{26}$$

Current and voltage measurements were also collected for the baselines. These were used to calculate the instantaneous power for each baseline according to Equation 27, then integrated to determine the energy used by the Terasic SoCKit board when it is running the simple program according to Equation 28. In these equations, the baseline power, voltage, and current replace "*SOC*" with "*base*" to indicate that they are baseline measurements. This energy is divided by the time for the baseline to determine a baseline energy usage rate $\dot{E}_{base\_TXR\_N}$, or baseline average power according to Equation 29. This baseline energy usage rate is multiplied by the average time for the program to which the baseline corresponds to determine the average baseline energy used by the Terasic SoCKit during the time in which the program was running according to Equation 30. At this point, the average energy used by the particular program running on the FPGA is determined by subtracting the average baseline energy from the average energy used by the entire Terasic SoCKit board

when the program was running on the FPGA according to Equation 31.

$$\vec{P}_{base\_TXR\_N} = \vec{V}_{base\_TXR\_N} \cdot \vec{I}_{base\_TXR\_N} \tag{27}$$

$$E_{base\_TXR\_N} = \sum_{m=0}^{M-1} \left[\left(\frac{\vec{P}_{base\_TXR\_N}(m) + \vec{P}_{base\_TXR\_N}(m+1)}{2} \right) \times \left(\vec{t}_{base\_TXR\_N}(m+1) - \vec{t}_{base\_TXR\_N}(m) \right) \right] \tag{28}$$

$$\dot{E}_{base\_TXR\_N} = \frac{E_{base\_TXR\_N}}{\vec{t}_{base\_TXR\_N}(M) - \vec{t}_{base\_TXR\_N}(1)} \tag{29}$$

$$\bar{E}_{base\_SOC\_TXR\_N} = \dot{E}_{base\_TXR\_N} \cdot (\bar{t}_{TXR\_N}) \tag{30}$$

$$\bar{E}_{TXR\_N} = \bar{E}_{SOC\_TXR\_N} - \bar{E}_{base\_SOC\_TXR\_N} \tag{31}$$

The results of this initial HITL simulation are fully discussed in Section 4.5.2.

4.5.2 First Attempt Results

Data for the first attempt at an Error Free HITL Simulation were collected following the procedures set forth in Section 4.5.1. Then, the data were processed according to Equations 23, 24, 25, 26, 27, 28, 29, 30, 31. Furthermore, the values of $\bar{E}_{TXR\_N}$ and $\bar{t}_{TXR\_N}$ are composed into vectors $\vec{\bar{E}}_{TXR}$ and $\vec{\bar{t}}_{TXR}$ which are plotted in Figure 17. Note that this figure plots the experimental results against the simulation results from Figure 11.

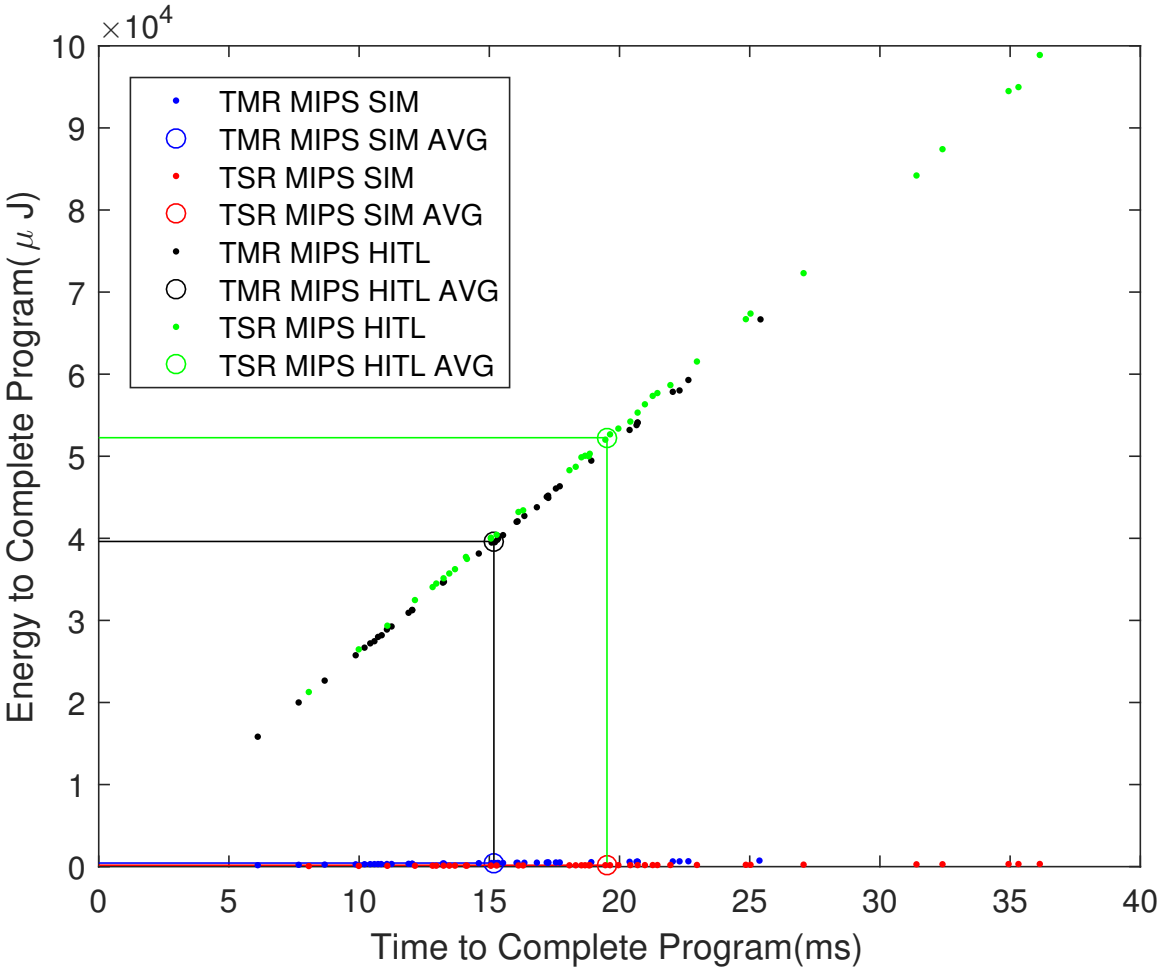

Figure 17. First HITL Attempt Energy vs. Time to Complete

Aside from the obvious discrepancy in energy, the software simulation timing predictions appear to match the HITL simulation results and meets the expectation that TSR MIPS programs would take longer to run than TMR MIPS programs. The discrepancy between the energy values predicted by the software simulations and the energy values recorded from the HITL simulation appears to be several orders of magnitude. Additionally, the energy used by TSR MIPS programs is significantly more than the energy used by TMR MIPS programs, which is contrary to the expectation that TMR MIPS would use more energy than TSR MIPS.

This discrepancy may be explained by the decision to combine the processor and

memulator onto a single SoCKit Development Board's Cyclone V FPGA. The energy measurements not only include the processor's energy usage, but also the memulator's energy usage. No accounting was made of memulator's power requirements when using the PowerPlay tool to estimate the FPGA's power requirements.

The violation of the expectation that TMR MIPS would use more energy than TSR MIPS may be explained by the fact that TSR MIPS programs are significantly larger than TMR MIPS programs. Not only do the TSR MIPS programs contain double the number of instructions as TMR MIPS programs, they also contain more temporary memory to temporarily store variables, additional instructions to create save/restore points, and additional instructions to recover from errors. The memory implemented in the TSR MIPS memulator is therefore more than twice the size of the memory implemented in the TMR MIPS memulator and would likely require at least twice as much power.

The Memulator consists of a large number of 32-bit registers (one for each instruction and needed memory location for a program) and surrounding combinational logic to store data to them and retrieve data from them. It would be reasonable to assume that these registers consume a significant amount of energy. It would also be reasonable to assume that a TSR MIPS Memulator would use more energy than an equivalent TMR MIPS Memulator because TSR MIPS programs not only double the number of instructions, but also add more instructions for creating save/restore points and error recovery. The logical conclusion of these assumptions is that the combination of TSR MIPS and TSR Memulator might consume more energy than the combination of TMR MIPS and TMR Memulator; however, to determine whether this is the case and whether the initial hypothesis was correct would require separating the Memulator from the processor so that the energy used by the processor can be measured separately.

To confirm these theories about the large energy discrepancy and expectation violation, the PowerPlay tool is used to analyze TMR MIPS and TSR MIPS when the memulators for those architectures are included on the same FPGA. While it is expected that the power required by the memulator is dependent upon the size of the program, only one program is used to confirm these theories. More specifically, one TMR MIPS program is implemented in a memulator with the TMR MIPS processor and the equivalent TSR MIPS program is implemented in a memulator with the TSR MIPS processor. This is done in order to quickly determine if these theories are correct because using Quartus to recompile 40 TMR MIPS programs with TMR MIPS processors and 40 TSR MIPS programs with TSR MIPS processors is a very time consuming task which is not attempted. If these theories do not prove true, not much time is wasted attempting to verify them.

The results of the revised PowerPlay analysis are shown in Table 11. The results of the updated analysis are used to compute simulated energy usage and the results are plotted in Figure 18

Table 11. PowerPlay Results for Processor and Memulator Together

Architecture	**ALMs**	**Registers**	**Dynamic Power (mW)**	**Variable**
TMR MIPS	8,845	9,270	46	$P_{TMR\ MIPS\ Mod}$
TSR MIPS	13,298	12,642	63	$P_{TSR\ MIPS\ Mod}$

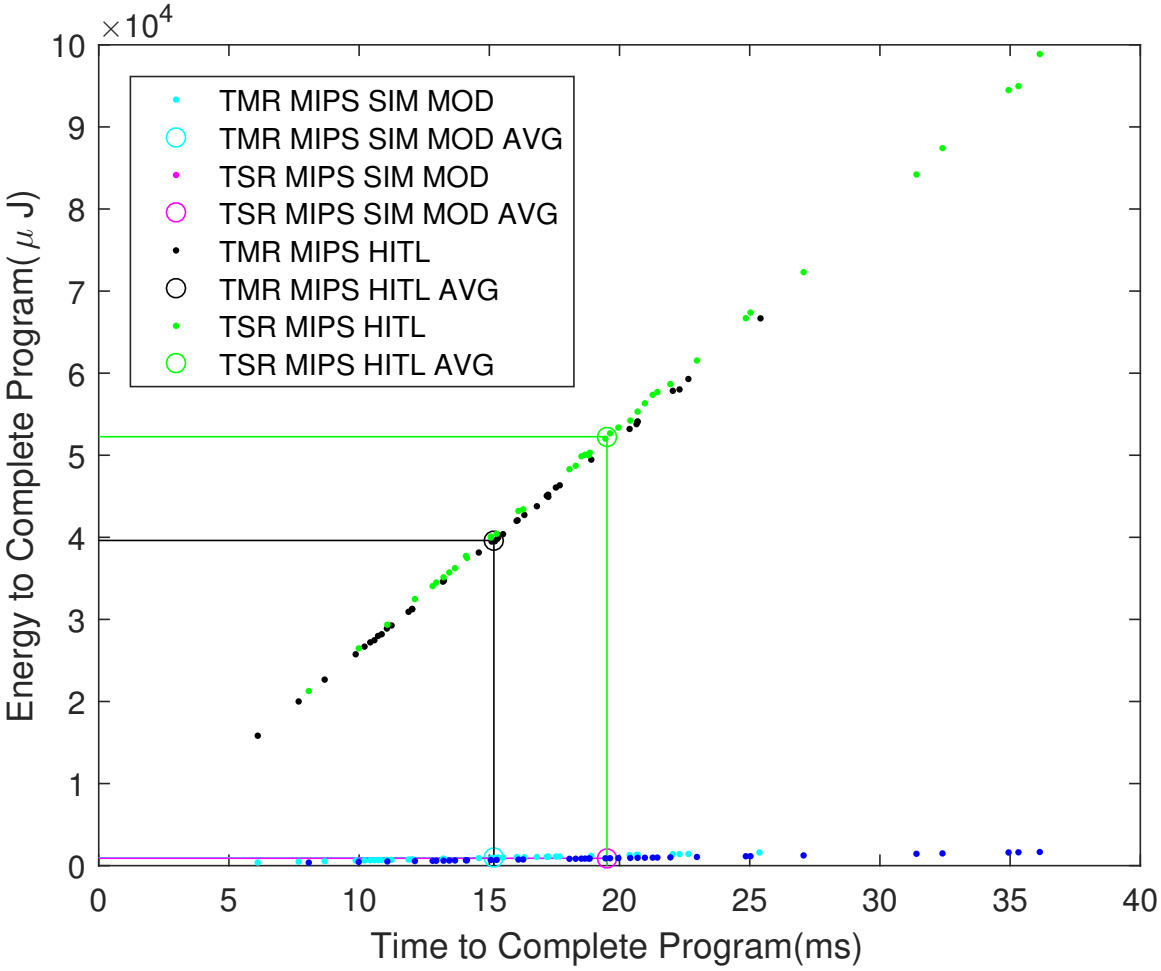

Figure 18. First HITL Attempt Energy vs. Time to Complete with Updated Energy Estimates

This updated power estimate clearly did not rectify the orders of magnitude difference between the simulation results and the measured results. This suggests that some other factor is affecting the HITL simulation results. One possibility is that there may be a problem with the method of collecting baseline energy usage and subtracting it from the energy used by the TMR MIPS or TSR MIPS processor and memulator. Another possibility might stem from static power consumption. TMR MIPS and TSR MIPS may consume different amounts of static power that were not accounted for in the baseline or in the PowerPlay tool. Static power was ignored because it was assumed that it was subtracted out when subtracting out the baseline.

Additionally, the PowerPlay tool showed no difference in static power for different architectures. It is also possible that the PowerPlay tool provided inaccurate power estimates. The ability of the PowerPlay tool to correctly estimate Cyclone V power usage was not independently verified or validated. More work will need to be done to investigate and address this issue.

Figure 18 did nothing to address the discrepancy of TSR MIPS consuming more energy than TMR MIPS. Figure 19 is an update of Figure 11 that shows the updated software simulation results along with the original software simulation results. This shows that both TMR MIPS and TSR MIPS use more energy when their memulators are included on the same FPGA, as expected, but TMR MIPS still uses more energy than TSR MIPS. The gap between TMR MIPS and TSR MIPS energy usage has been greatly reduced, but still does not address the fact that the HITL simulations show TSR MIPS using considerably more energy than TMR MIPS. Once again, other factors are affecting the HITL results that are not accounted for in software simulations.

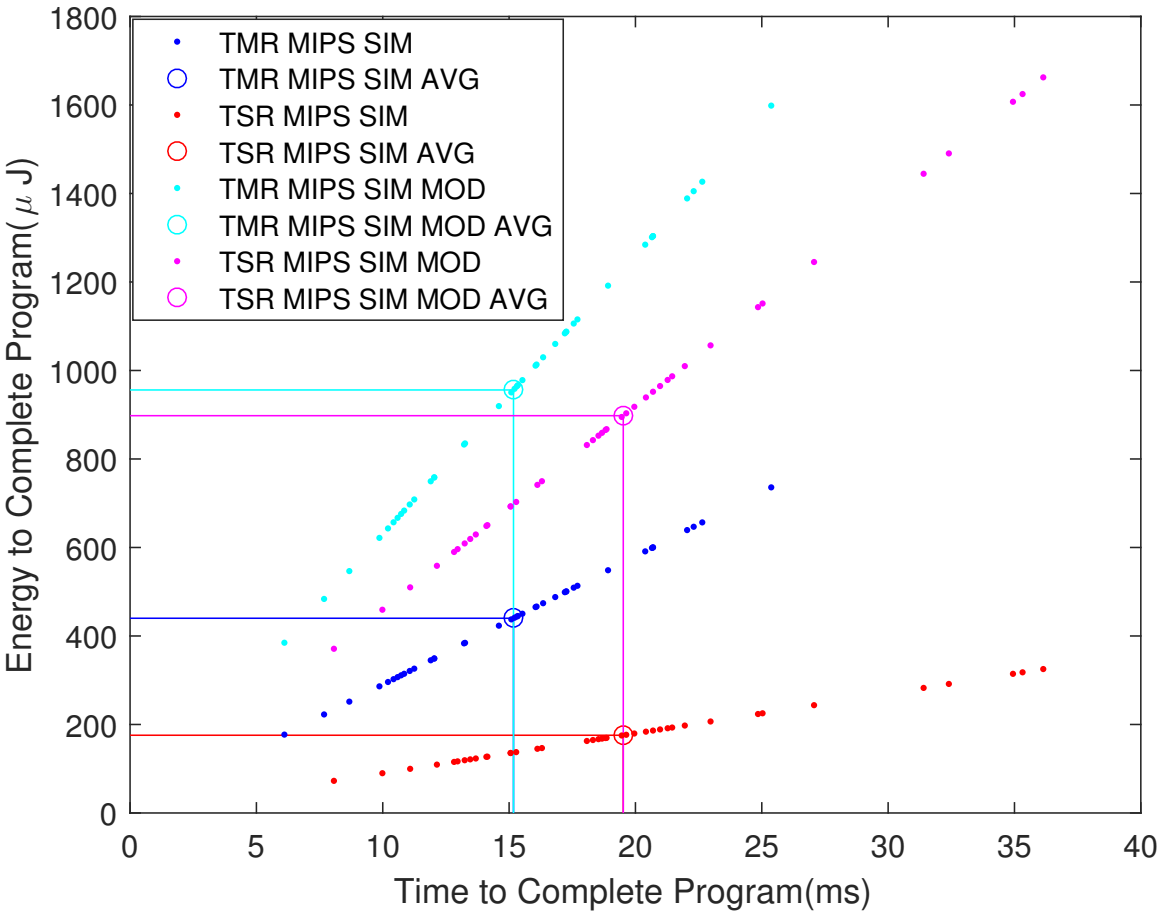

Figure 19. First HITL Attempt Software Simulation Energy with Updated Energy Estimates

4.5.3 Second Attempt Methodology

The second attempt at error free HITL simulation is setup so that one Intel Cyclone V FPGA on a Terasic DE10-Standard board is programmed to run a processor and a separate Intel Cyclone V FPGA on another Terasic DE10-Standard board is programmed to run the Memulator. This setup is designed to enable measurement of the energy used by the processor alone. The processor can be Basic MIPS, TMR MIPS, TSR MIPS (which is Basic MIPS), or AHR MIPS and the Memulator is one that is appropriate for the associated processor on the first FPGA; namely a Basic MIPS program, TMR MIPS program (which is the Basic MIPS program), TSR MIPS

program, or AHR MIPS program respectively. The two DE10-Standard boards are connected using the on board general purpose input-output (GPIO) 40-pin header and two GPIO 40-pin headers on a Terasic HSTC to GPIO Daughter Board connected to each DE10-Standard. Of the 120 pins available on the three GPIO 40-pin header connections, 32 are used for the address signal, 32 for data to write to memory, 32 for data to read from memory, 1 for the read enable signal, 1 for the write enable signal, 1 for memory ready signal, and 1 for done signal upon program completion; this is a combined total of 100 signals. Additional pins are used to establish a common ground between the two boards. For full details on the interconnections between the two DE10-Standard boards, please refer to the Air Force Institute of Technology technical report on DE10 Pins and Other Connections which can be made available upon request [37].

In addition to the signals above, a UART error signal, the o_DONE signal, and an additional ground are exported from individual pins on the daughter board attached to the DE10-Standard board running the Memulator as described in the DE10 Pins and Other Connections AFIT technical report. The UART error signal is used to monitor whether the processor has encountered an error. The o_DONE signal is monitored by an oscilloscope to determine the end of the program running on the DE10-Standard board running the processor as well as the restarting of the program. The ground signal is used as a reference point for the UART error signal.

After connecting all the wires between the boards, each connection is tested with a continuity meter to ensure that all connections were successfully made. These measurements are made by touching one lead of the continuity meter to one of the GPIO pins solder beads on the back of the DE10-Standard or daughter card and the other lead to the connected pins solder beads on the back of the other DE10-Standard or daughter card. All paired pins must demonstrate successful continuity

in order to conduct HITL simulations or the processor and Memulator will not be able to communicate with one another.

The UART error signal sends UTF-8 characters to indicate various error codes associated with TMR and TSR MIPS processors. These error codes are generated by the Memulator. In addition to being designed to function as a memory, the Memulator has also been programmed to determine if any control flow errors occur. For example, the Memulator knows that if the previous instruction was not a load word, store word, or branch instruction, that the next read request from the processor will be for the next instruction in memory (i.e. next address = last address + 4). If this is not the case, the Memulator signals an error. If the last instruction was a load word, the Memulator calculates the address from which the processor should read and signals an error if that address is not read. If the last instruction was a store word, the Memulator calculates the address to which the processor should write and signals an error if the processor reads instead of writing or attempts to write to the wrong address. If the last instruction was a branch instruction, the Memulator calculates the next instruction address and the branch address. The Memulator only signals an error if the processor does not attempt to read from the next instruction address or branch address because the Memulator has no way of knowing whether the processor should take the branch or not. The Memulator can also determine when TMR or TSR MIPS are attempting to recover from an error, or when the processors have failed to recover from an error. In the case of TMR MIPS, the Memulator can determine whether TMR has encountered a single processor error or multiple processor error. The error signals are recorded by a laptop. A complete summary of the error codes are provided in Table 12.

Table 12. Memulator Error Codes

Code	Architecture	Description
A	TMR/TSR	Expecting write after SW but received read request
B	TMR/TSR	Expecting read after LW but reading wrong address
C	TMR/TSR	Branch to incorrect location
D	TMR/TSR	Last instruction not LW, SW, or branch. Reading next instruction from wrong address. PC $\neq$ PC+4
E	TMR/TSR	End of program incorrectly reached
F	TMR/TSR	Attempt to read from out of bounds memory location
G	TMR/TSR	Writing word to wrong address after SW
H	TMR/TSR	Unexpected write to memory when last instruction not SW
I	TMR/TSR	Attempt to write to out of bounds memory location
J	TMR	TMR timeout reached - TMR may be attempting Type A error recovery
K	TMR	Read word from save/restore point in memory. TMR Type B error recovery in progress
L	TSR	TSR error recovery started
M	TSR	Error recovery code entered by failed return from save/restore point creation
N	TMR/TSR	No error
S	TMR	Creating save/restore point
X	TMR/TSR	Timeout exceeded - failed to read from memory. Prcoessor locked up or failed

Current and voltage measurements are made the same way as in the first attempt and are illustrated in Figure 14. The overall experimental setup is shown in Figure 20.

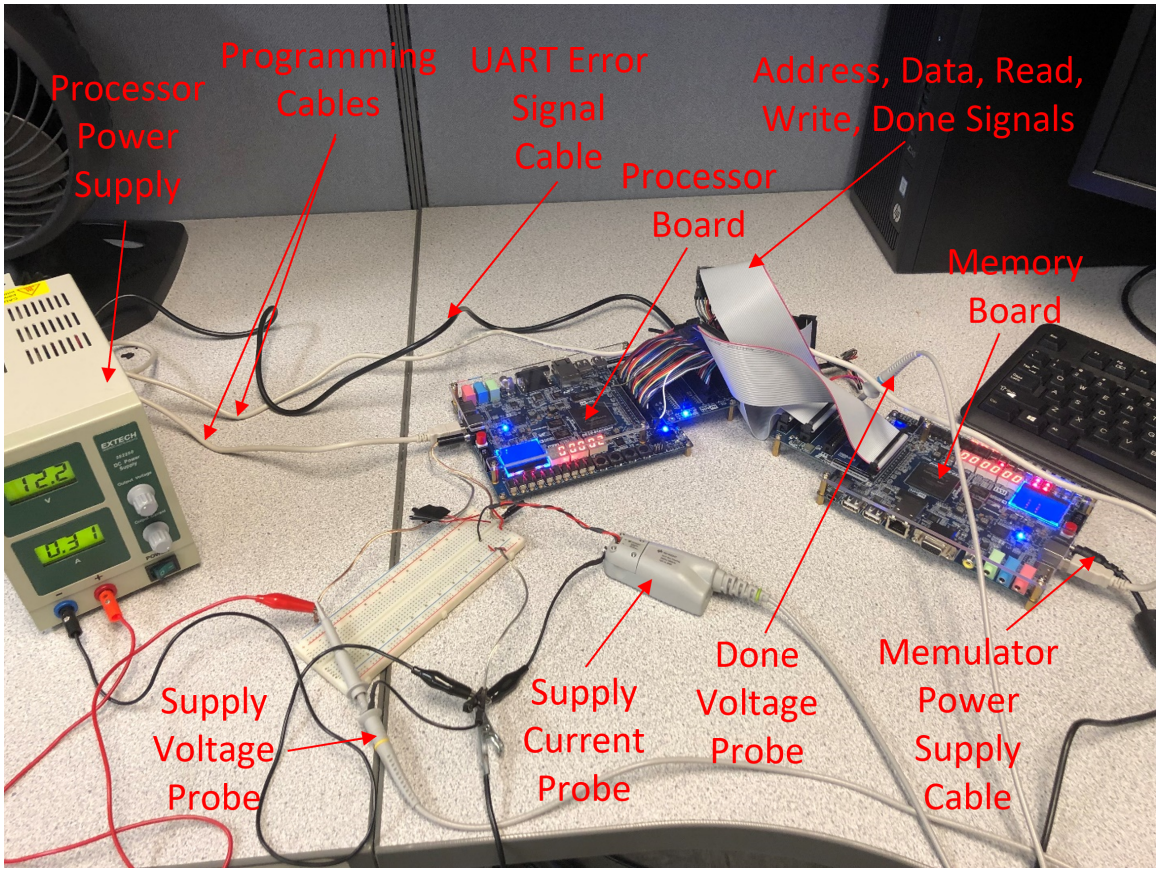

Figure 20. HITL Attempt 2 Experimental Setup

Timing measurements are also made for the second attempt in the same manner they were made for the first attempt. The current, voltage, and timing data are also used to calculate energy usage as before. Baseline data is recorded prior to each Basic MIPS, TMR MIPS, TSR MIPS, and AHR MIPS data collect. As before, each data collect consists of capturing five program completions on the oscilloscope and doing that twice for each program so that 10 program completions are recorded. The baseline energy is again subtracted from the energy used by the program running on the FPGA to determine the energy used only by the specific architecture running the program.

Section 4.5.4 discusses the results of the second HITL attempt.

4.5.4 Second Attempt Results

Learning from the first attempt that implementing a processor and its memulator on the same FPGA confounds any sort of meaningful energy analysis, the second attempt at a HITL simulation placed a processor on one FPGA and its memulator on a separate FPGA. Unfortunately, the various MIPS processors failed to function properly when operating in this fashion. This problem was explored in detail for the TMR MIPS processor.

The first step in the troubleshooting process was to use the light emitting diodes (LEDs) and hex displays on the processor and memulator DE-10 Standard boards to display meaningful signal and state information. These visual indicators revealed that TMR MIPS attempts to read the first instruction from the memulator. The memulator then provides data back to TMR MIPS and TMR MIPS appears to receive that data. At this point, the TMR Voter fails to transition to the next state in which it should provide the results from memory back to the three Basic MIPS processors. Instead, it stays in the state where it continues to request the first instruction from memory.

The next step was to implement a clock divider on both FPGAs. The clock divider on each FPGA could be varied by adjusting the switches on each DE-10 Standard board. The clock dividers were independent of one another and only set the clock speed for the FPGA on the same board. Reducing the clock speed to approximately 2 Hz on both FPGAs caused the TMR MIPS processor to function correctly. Increasing the clock frequency on both boards would allow TMR MIPS to function properly for a while, but would then encounter a Type A or Type B error when no errors were being injected. Occasionally, the TMR MIPS processor would freeze up as observed in the first troubleshooting step, but the processor would stop at different TMR Voter and Basic MIPS processor states than the one observed in the first step. Operating

TMR MIPS at such a low frequency would prevent any meaningful measurements of program runtime and energy usage, so more troubleshooting steps were pursued.

The third troubleshooting step was to use Quartus II to perform a timing analysis on the TMR MIPS processor and TMR Memulator. These timing analyses showed that neither could run at the default 50MHz clock frequency, but could run at 25MHz. A clock divider that reduced the clock frequency to 25MHz was implemented on the processor and memulator and Quartus II indicated that the updated designs passed the timing analysis. When implemented in hardware, the results were identical to what was revealed in the first troubleshooting step.

The final troubleshooting step built upon the third troubleshooting step by utilizing the Quartus II SignalTap tool to monitor the signals on the TMR MIPS processor and TMR Memulator while in operation. The results of the SignalTap analysis were identical to the results of the first troubleshooting step once again.

The results of this troubleshooting process suggest that there is a problem with how Quartus II implemented the TMR MIPS processor design in hardware that was somehow partially corrected when the switched clock divider was implemented. It is beyond the scope of this research to delve into the inner workings of Quartus II and discover how it makes its hardware implementation decisions. Future work may implement the TMR Voter in the same manner as the Basic MIPS processor; the TMR Voter may be implemented entirely from NAND gates and D-flip-flops rather than using high-level VHDL language to describe a state machine. This approach may force Quartus II to implement the hardware such that the TMR Voter does not fail to make a state transition after receiving a response from memory.

4.6 Summary

This chapter discussed the approach to verifying that Basic MIPS (unmitigated processor) TMR MIPS (TMR strategy), TSR MIPS (TSR strategy), and AHR MIPS (AHR strategy) worked as designed. It further examined the method whereby the architectures could be compared to one another in terms of program runtimes and energy usage in the absence of errors so that useful determinations could be made about the relative advantages and disadvantages of each architecture with the goal of highlighting the advantage of AHR over TMR or TSR alone. It was shown that, in the absence of errors, AHR does combine two different redundancy methods, allows switching between these redundancy methods, provides flexibility in selecting when to switch between these redundancy methods, and opens up tradespace in time and energy allowing AHR to function more like TMR or TSR or anywhere in between in terms of time and energy performance.

This chapter also discussed how to perform Hardware-in-the-Loop (HITL) simulations in order to collect representative data on the performance of these architectures when implemented in hardware.

These simulations, analyses, and HITL simulations have not yet demonstrated the ability of AHR to operate in radiation environment where SEUs and SETs may occur. Chapter V will examine how to inject errors into TMR, TSR, and AHR MIPS for the purpose of evaluating their performance in the presence of errors in Chapter VI.

V. Error Injection Development

5.1 Introduction

The software simulations, analyses, and hardware-in-the-loop (HITL) simulations in the previous chapter established a baseline of performance for each of the architectures and provide a basis of comparison between them; however, the previous chapter did not indicate how the architectures will behave in the presence of errors. Error injection is the next step toward evaluating the architectures when Single Event Upsets (SEUs) and Single Event Transients (SETs) are present, but one must first determine the rate at which errors are expected to occur. Section 5.2 discusses the method used to determine the appropriate error rate. Then, Section 5.3 discusses the architecture needed to inject errors. Section 5.4 provides the analysis tools that will be used to evaluate the performance of TMR, TSR, and AHR MIPS with regards to time and energy when errors are injected.

5.2 Error Rate Determination

Section 2.4 previously introduced the concept of radiation comparisons and the idea of comparing the radiation testing performed in this research to that of previous experiments by Normand et al. [67, 66] and the Cibola flight experiment [108]. While a number of radiation tests were devised to probe the vulnerability of the Cyclone V FPGA in Section 5.2.1, the results of the neutron testing in Section 5.2.2.1 proved the most useful in determining an error rate for error injection purposes.

5.2.1 Radiation Testing

Radiation testing is used to determine the vulnerability of the Intel Cyclone V FPGA to SEUs and SETs in terms of upset rate or failures-in-time (FIT). This

information is key in determining how much mitigation TMR, TSR, and AHR MIPS provide as well as how frequently errors should be injected into these architectures in simulations and analyses. Section 5.2.1.1 discusses the checkerboard pattern used to test the Cyclone V's vulnerability. Section 5.2.1.2 discusses the test methods used for neutron, carbon ion, and proton radiation experiments. The section concludes by discussing the limitations of these experiments in Section 5.2.1.3.

5.2.1.1 Checkerboard Implementation

Implementing a static checkerboard pattern directly in the Cyclone V's user logic was not possible because the Intel Quartus II Version 14.1 synthesizer replaced the registers with constants. To overcome this problem, each register had its output fed back to the input through an inverter. This feedback loop caused the checkerboard pattern to alternate on every clock cycle. The alternating checkerboard pattern made it impossible to compare the output of each register to a fixed value; therefore, each register was paired with a complementary register storing the opposite value. The outputs of the two registers become the inputs to an exclusive-nor (XNOR) gate. The output of the XNOR gate is 0 when the two registers differ in value and 1 when they have the same value. The two registers can only have the same value when an error has occurred. Note that if both registers have an error, the output of the XNOR could still be 0; however, this is expected to be a rare event. The occurrence rate of this rare event is considered to be insignificant because this experiment was intended to discover a relative level of vulnerability to radiation rather than a precise vulnerability estimate. Figure 21 shows how each checkerboard memory location is formed.

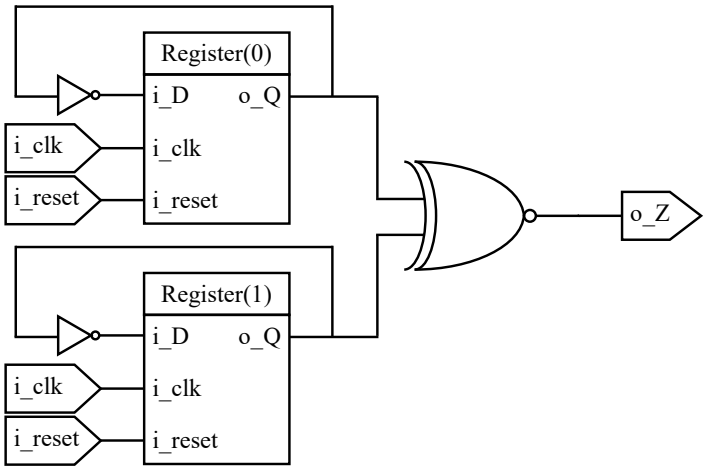

Figure 21. Checkerboard Location Register Pair with XNOR Gate

Outputs of all checkerboard locations are subsequently added to determine the total number of errors in the checkerboard pattern. These additions are carried out by a network of carry select adders. The XNOR outputs of the first three checkerboard locations (0, 1, and 2) are supplied as the A, B, and carry-in inputs of a 1-bit carry select adder. The outputs of the next three checkerboard locations (3, 4, and 5) are supplied as the A, B, and carry-in inputs of another 1-bit carry select adder. The sum and carry-out outputs of each 1-bit carry select adder are concatenated to form a single 2-bit sum that ranges in value from 0 to 3. These 2-bit sums are supplied to the A and B inputs of a 2-bit carry select adder. The carry-in of the 2-bit carry select adder is from checkerboard location 6. The sum and carry-out outputs of each 2-bit carry select adder are concatenated to form a single 3-bit sum that ranges between 0 and 7. At this point, a pattern is beginning to form. An n-bit adder has two n-bit inputs and a carry-in. The sum and carry-out of the n-bit adder are concatenated to form a single n+1-bit sum; that ranges between 0 and $2^{n+1} - 1$. The overall sum for 2^N bits requires the last adder have two N bit inputs and a carry-in input, and the final sum has $N + 1$ bits; however, the maximum value is still limited to 2^N because 2^N bits are summed together. Figure 22 shows a small subset of the adder network.

It depicts how the first 15 checkerboard memory locations' error signals are added together. The pattern for the adder network is readily discernible from this subset.

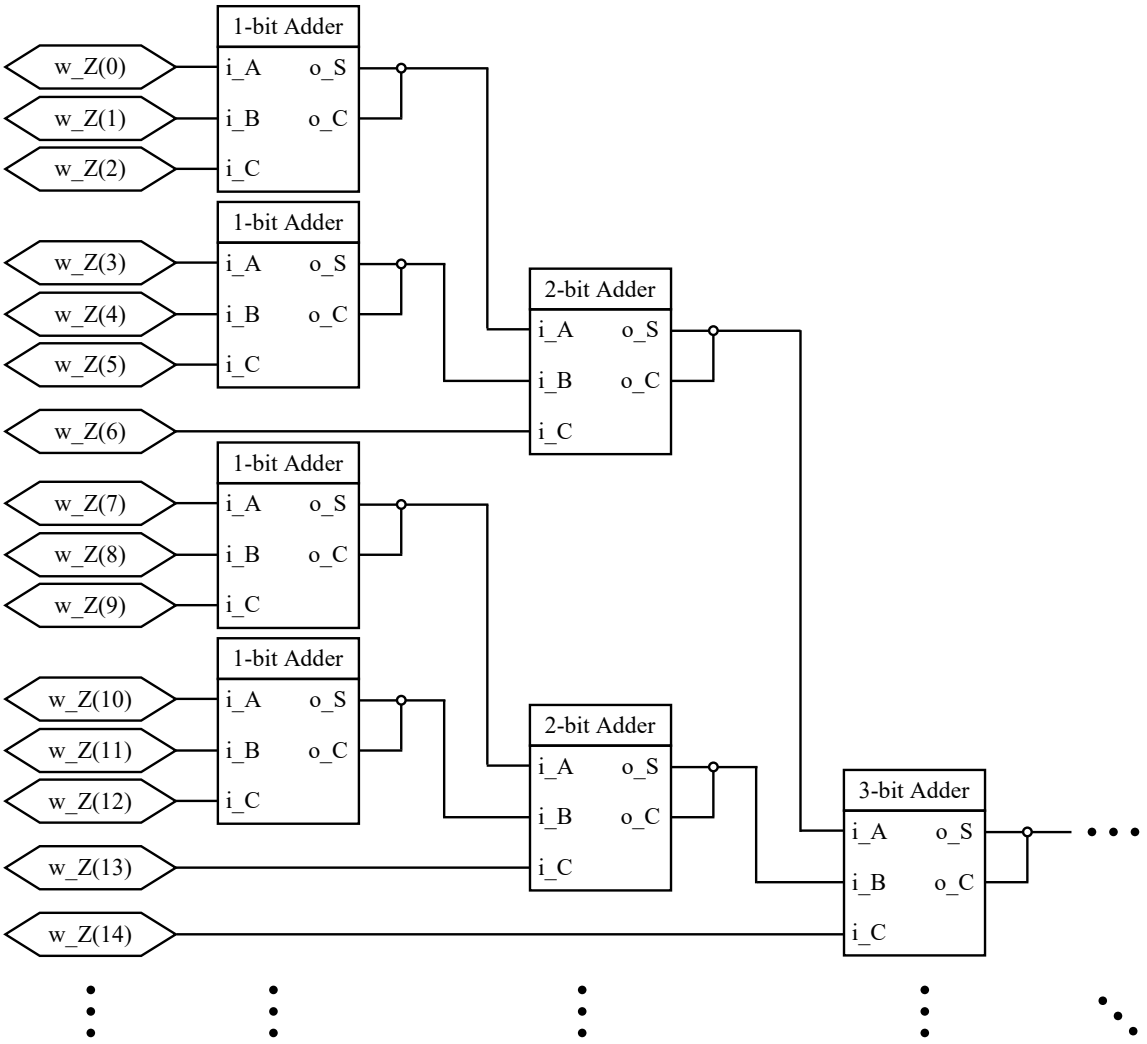

Figure 22. Subset of Checkerboard Adder Network Showing First Three Levels to Add Error Signals from the First 15 Memory Locations

Due to FPGA routing constraints, the maximum value for N was $N = 13$. Quartus II failed to converge on a solution to fit a design with $N >= 14$ on the FPGA. The largest possible value for N was selected in order to maximize the amount of vulnerable configuration memory and user logic on the FPGA, which increases the

likelihood that SEUs and SETs will be observable during testing.

In addition to the checkerboard pattern and associated adders, a buffer and serial transmitter were created to export the data during testing. The buffer detects every time a change in the error count occurs. If the error buffer and serial transmitter are idle when a change occurs, the error buffer provides the least significant 8 bits of the error count to the serial transmitter for transmission followed by the UTF-8 comma character as a delimiter. The serial transmitter transmits data at 11,520 bytes per second. The transmitted data is recorded on a computer running Matlab.

The checkerboard pattern, buffer, and serial transmitter used 20,980 of the Cyclone V's 41,910 adaptive logic modules (ALMs). Each ALM has eight inputs, an adaptive look-up table (LUT), and four registers [44]. A total of 16,437 of the Cyclone V's 166,036 user registers were used; 2^{14} were used by the checkerboard and 53 by the buffer and serial transmitter. The DE10-Standard drew between 0.36A and 0.38A when not operating in a radiation environment. This current range is the DE10-Standard's nominal current. When the DE10-Standard current exceeds this range, it is interpreted as a configuration memory SEU causing a short circuit between two signals.

5.2.1.2 Testing

The Cyclone V implementing the checkerboard pattern was exposed to neutron, carbon ion, and proton radiation. Carbon ion and proton tests required the Cyclone V chip to be de-lidded. De-lidding is the process of removing packaging material from a chip and is commonly done for ion tests so that the ion beams can produce SEUs and SETs in the electronic device being tested. Without de-lidding, the ions used in testing would lack sufficient energy to penetrate the packaging material and chip silicon to produce SEUs and SETs in the electronic device. Ions in space have more

energy and are able to penetrate packaging materials [51]. The Cyclone V is not a flip-chip design, so the carbon ion and proton beams had to penetrate the metal layers to produce SEUs and SETs in the memory cells and combinational logic. (De-lidding is unnecessary for neutron testing as neutrons readily pass through packaging materials with little impact upon their ability to cause SEUs and SETs in the electronic device.)

During the course of testing, there were several instances where the Cyclone V stopped transmitting data and the current draw increased. Whenever this occurred, the DE10-Standard board had to be reset, reprogrammed, and checked for functionality before testing could resume. Each time the DE10-Standard was reset and reprogrammed, it passed functional tests and no permanent device failures occurred. The DE10-Standard was also reset and reprogrammed if the error count reached 255, the maximum value that can be reported (see Section 5.2.1.3 for more details).

Testing was conducted at Sandia National Laboratories (SNL) Ion Beam Laboratory (IBL). Neutron testing was conducted using the IBL's 350 kV High Voltage Engineering Europa (HVEE) Implanter with a D-T neutron source producing 14MeV neutrons. Carbon ion and proton experiments were performed with the High Voltage Engineering (HVE) 6 MV Tandem accelerator with DE10-Standard boards placed in the end station of the Qualification Alternative to the Sandia Pulse Reactor 3 (QASPR-3) beam line. The DE10-Standard boards were mounted in the QASPR-3 chamber so that the text on the Cyclone V chip was right-side-up. Prior to each carbon ion or proton test, the board was reset and reprogrammed before closing the testing chamber. The chamber was then evacuated so that the vacuum pressure was on the order of 10^{-5} torr. The carbon ion and proton beam spots were square, but much smaller than the Cyclone V chip; therefore, multiple beam shots were fired and the DE10-Standard moved between shots to completely irradiate the Cyclone V. These shots were performed in overlapping squares on a grid pattern to ensure the

entire chip was irradiated to the same average fluence levels. Three different DE10-Standard boards, each with its own Cyclone V FPGA, were used in the tests; one board was used for each radiation type (one for all neutron tests, one for all carbon ion tests, and one for all proton tests).

Neutron Testing The DE10-Standard was placed in direct contact with the canister housing the D-T neutron source and positioned so that the Cyclone V was in the same horizontal plane as the beam striking the target to maximize neutron fluence on the Cyclone V. The Cyclone V was approximately 6mm from the canister's outer wall. The neutron source was continuously active for one week with only a few dropouts.

During these tests, Matlab was set to record data in 15 minute intervals. At the end of each 15 minute interval, the data was saved before recording data for another 15 minutes. This was done continuously, with few exceptions, from before the D-T source was activated until after it was deactivated. Baseline data was collected prior to D-T source activation and after deactivation to ensure no errors were detected in the absence of radiation.

Carbon Ion Testing The carbon ions used in this testing were 35MeV C^{6+} ions. A total of three tests were performed. The average fluence for each test was 9.693×10^7, 1.097×10^7, and 9.834×10^5. It was anticipated that the first test would not produce a large number of errors and that the second and third tests would increase the average fluence from one test to the next with a corresponding increase in the number of errors, but that was not the case. Because the number of errors was so large after the first test, the fluence was decreased in an attempt to decrease the number of errors on the last two tests.

The beam fluence was not uniform such that highest fluence levels were in the

shape of a circle in the middle of the square and there was a significant decrease in fluence at the edges and corners of the square. This required significant overlap of the beam between adjacent shots, so the DE10-Standard was moved in a 5x5 grid pattern.

During these tests, Matlab was set to record a few seconds worth of data starting just before each beam shot was fired and ending just after each beam shot was fired for each carbon ion test. As a result, up to 25 data sets were recorded for each carbon ion test; one data set was collected for each grid location. These data sets were numbered starting with 0 increasing to 24.

Proton Testing The IBL's QASPR3 beam line generated 4.5MeV protons for these tests. The beam fluence was uniform with some decrease in fluence along the edges of the square. This required minimal overlap of the beam between adjacent shots, so the DE10-Standard was moved in a 4x5 grid pattern. A total of three tests were performed. The average fluence for each test was 9.454×10^9, 9.939×10^{10}, and 9.173×10^{12}.

During these tests, Matlab was set to record data for the entire length of time over which the beam shots would be fired. However, during the third test, the beam shots took longer than anticipated and Matlab stopped recording as the predefined timeout was exceeded. Between beam shots, Matlab recording was restarted. The timeout period was exceeded again before testing was completed. In total, four data sets were collected during the third test.

5.2.1.3 Experiment Limitations

Carbon ion and proton beam testing could not target specific gates, transistors, or active regions of the FPGA because the locations of those elements are proprietary. Reverse engineering the Cyclone V to determine the locations of these elements is out

of scope for this research.

The maximum error count that can be recorded is 255 even though the true error count may be higher because only the least significant 8 bits of the error count are transmitted. When the serial transmitter repeatedly sends an error count of 255, this indicates that the error count is constantly changing, but that the value of the error count is always at or above 255. These changes are no longer observable and add no value to the results. The decision to only use the least significant 8-bits was made for two reasons. First, it was incorrectly assumed that the error count would not be higher than 255 during the testing time periods of interest. Second, the data rate is maximized by only transmitting the least significant 8-bits. Maximizing data rate was seen as a method to capture errors as soon as they occurred. A lower data rate might allow multiple errors to occur without logging between transmitting periods.

Another decision made to maximize the data rate was to not implement handshaking between the serial transmitter and Matlab. Handshaking would also have required additional circuitry on the FPGA that would have been vulnerable to radiation induced errors. This vulnerability might have caused the serial transmitter to fail if it could not respond to handshaking signals appropriately. Without handshaking, there was no way to timestamp individual data transmissions because Matlab samples and stores the incoming data continuously and is only able to track the time at which the data collection started and finished. For example, if 15 minutes of data were collected when Matlab was set to collect 15 minutes of data, then it would be possible to determine when every single byte was received during that time period. However, if only 5 minutes of data were received, it would be impossible to determine when each byte was received. This ambiguity in data timestamping is also a result of the decision to only have the error count transmitted when the error count changes rather than transmitting the error count continuously.

The second problem arising from the lack of handshaking is that Matlab could lose synchronization with the FPGA serial transmitter. If the transmitter is transmitting data before and during the time period when Matlab data collection begins, it is possible for Matlab to view any falling transition as the start bit of the serial data transmission. In the best-case scenario, this would occur at the correct start bit. In the second-best-case scenario, it would occur at one of two falling transitions in the middle of transmitting the comma character which is used as a delimiter. In this second-best-case scenario, the start of the comma is received as the last few bits of one byte and the end of the comma is received as the first few bits of the next byte. The start and stop bits transmitted by the serial transmitter are received in the middle of every byte received by Matlab. Data can be extracted by searching for these broken up commas and determining data to be in between them. Unfortunately, because two of the data bits will be interpreted as start and stop bits, only six bits of data are recoverable and the remaining two bits could cause the data to have one of four values. In the worst-case scenario, Matlab would detect the transition in the middle of a byte of data rather than a comma. In this scenario, commas would be impossible to locate and data would be impossible to recover. All three of these scenarios were observed during the experiment.

5.2.2 Radiation Testing Results and Analysis

5.2.2.1 Neutron Testing

During the neutron testing period, 655 data sets of 15 minutes each were collected. During this time period, there were 23 continuous data collections between interruptions where data transmission stopped and current draw increased. The shortest collections were individual 15 minute data sets while the longest collection contained 39, 15 minute data sets. The data collected for the longest continuous collection is

plotted in Figure 23. The plot includes dividers to differentiate between individual data sets which range from set number 538 to 576. Error counts of -10 are used to indicate data was not transmitted or unrecoverable. Plotting every single data point was impossible due to the large amount of data recovered. Instead, envelopes of the error count are plotted. When the error count alternates between two values, the plot shows the maximum and minimum values. There are still some instances where the maximum and minimum values fluctuate rapidly. During some data sets, the plot splits into four different colors where data was not received synchronously, but was recoverable. These colors indicate the four different values the data could be. In this plot, it appears that the correct error count is indicated by the red data every time the data was recovered from out of sync data. When the only visible data is blue, that means that the data was collected synchronously and the all possible lower bound values are identical to one another and all possible upper bound values are identical to one another. The last thing to note about this figure is that the data appears to show a linear increase in error count over time.

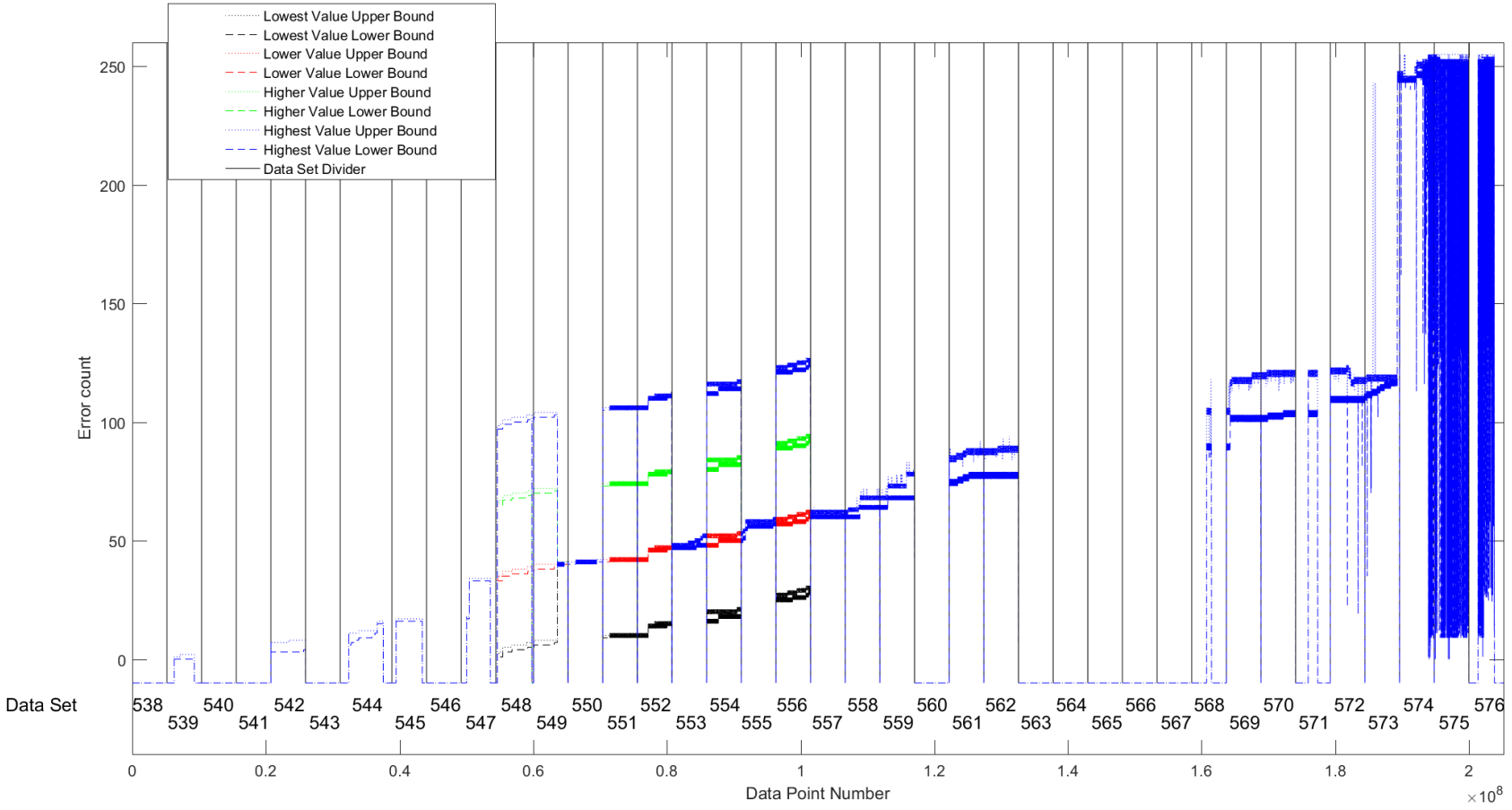

Figure 23. Neutron Test 538-576

Figure 24 shows a plot of the FIT rates from each continuous data collection. The FIT rates were calculated by dividing the error count at the last "stable" point in the continuous data set by the time over which the data was collected. The time period used is the stop time of the last data set in which the data appeared to be "stable" minus the start time of the first set. The stop time used does not match the true stop time due to the ambiguity in the data's time of arrival because the lack of handshaking between the FPGA and the computer. The upper bound FIT rate is taken to be the maximum value of the upper bound where the last "stable" point appears divided by time in hours, then multiplied by 10^9 to convert from error rate to FIT rate. The lower bound similarly uses the minimum value of the lower bound where the last "stable" point appears. In Figure 23, the last "stable" point is near the end of set 573. The FIT rate plot in Figure 24 appears to follow the trend of decreasing neutron fluence over time as the neutron source is depleted. The neutron production rate is shown in Figure 25 where the time scale shows the time in hours from the time the neutron source was activated. The two plots viewed together show that the number of neutrons being produced appears to have a direct impact on the FIT rate as both plots show a downward trend.

The unmitigated Cyclone V FIT rates shown are roughly 10^{10} or higher. In order to compare this rate to the Cibola rate, a comparison of the two fluence levels is in order. First, the neutron production rate is assumed to be $10^9/sec$ because the production rate is greater than or equal to $10^9/sec$ as shown in Figure 25. The neutron flux is then given by

$$\phi = \frac{PR}{4\pi r^2} = \frac{10^9}{4\pi(5cm)^2 \cdot sec} \approx 1.15 \times 10^{10} \frac{n}{cm^2 \cdot hr} \tag{32}$$

This flux is approximately ten times greater than the neutron flux in the Normand et al. experiments [67, 66] and indicates that the FIT rate seen here is ten times

greater than the FIT rate that would have been seen if exposed to the same neutron environment as the Normand experiments; therefore, the neutron experiment FIT rate should be divided by ten to be comparable to the Cibola Flight Experiment FIT rate. The Cibola Flight Experiment FIT rate was 3.25×10^7 and the neutron experiments adjusted FIT rate is approximately 10^9 or higher. The Cibola Flight Experiment experienced 0.78 SEUs/device/day and the Cyclone V experienced approximately 240 SEUs/device/day before the neutron flux adjustment and 24 SEUs/device/day after the neutron flux adjustment. This error rate is equivalent to an unadjusted rate of 10 SEUs/hour ($\frac{1}{6}$ SEUs/min, $\frac{1}{360}$ SEUs/s) or an adjusted rate of 1 SEU/hour ($\frac{1}{60}$ SEUs/min, $\frac{1}{3600}$ SEUs/s). One additional assumption in making this comparison is that the monoenergetic 14 MeV neutrons in this experiment are comparable to the broad spectrum neutrons in the Normand et al. experiments. Another assumption is that the Xilinx FPGAs used in the Cibola mission will experience the same upset rates as the Cyclone V used in this experiment. Finally, after making all of these assumptions, all of which are questionable, and having a desire to develop a rough, order of magnitude comparison between the neutron test results presented here and the Cibola mission, the neutron experiment FIT rate is approximately one hundred times greater than the mitigated FIT rate seen in the Cibola Flight Experiment. This FIT rate is unacceptable in space, which underscores the necessity of radiation mitigation for FPGAs. Additionally, since most of the programs in Section 4.4 completed in 50ms or less, it would be expected to see an error once in approximately every 72,000 program runs based on an average upset rate of 1 SEU/hour. This further bolsters the results of error free software simulations since it is far more likely that a program will complete without experiencing an SEU than a program will experience an SEU while running. It also provides assurances that the switching capability and flexibility demonstrated by AHR in error free simulation and analysis will be realized

in implementation because errors are so infrequent.

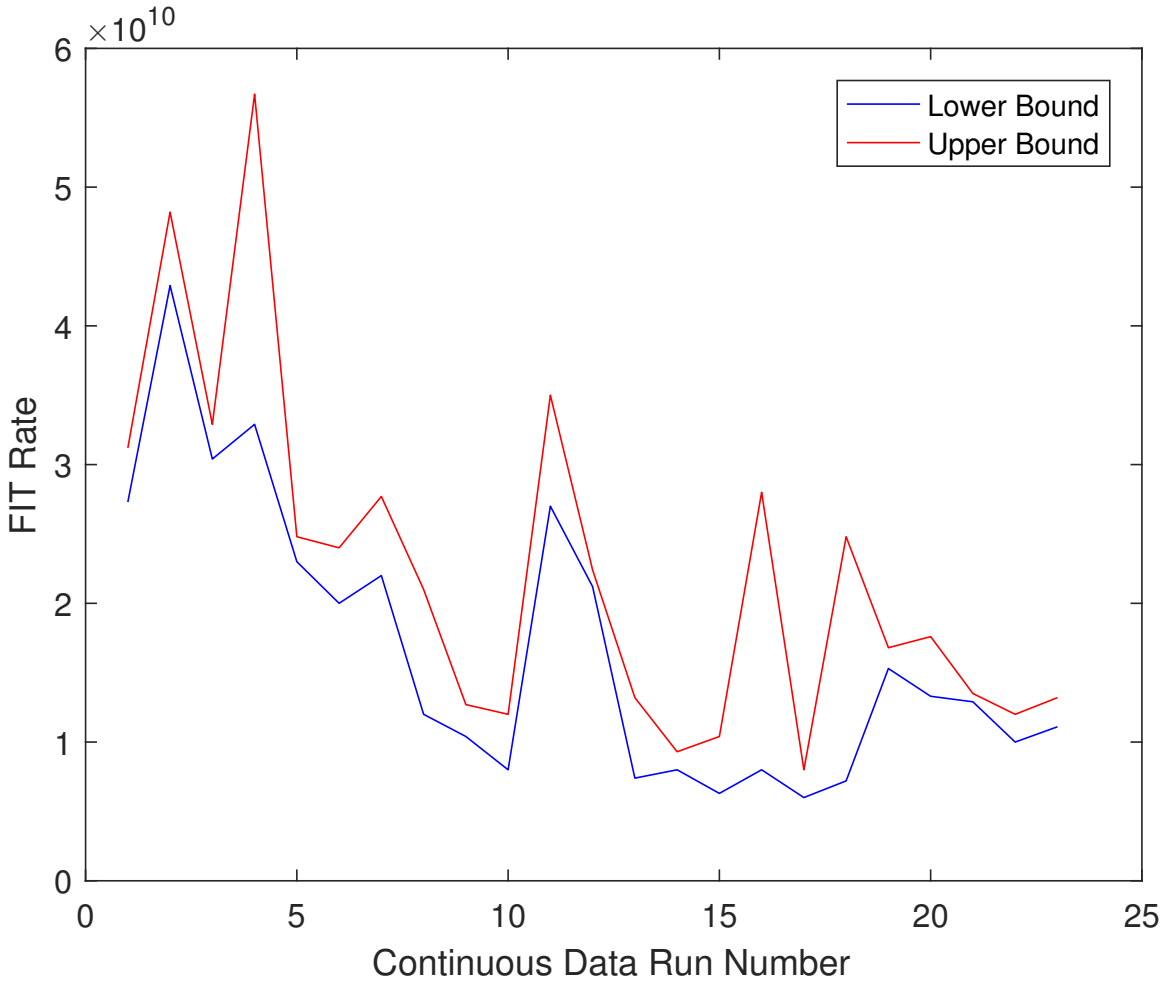

Figure 24. Neutron Test FIT Rates

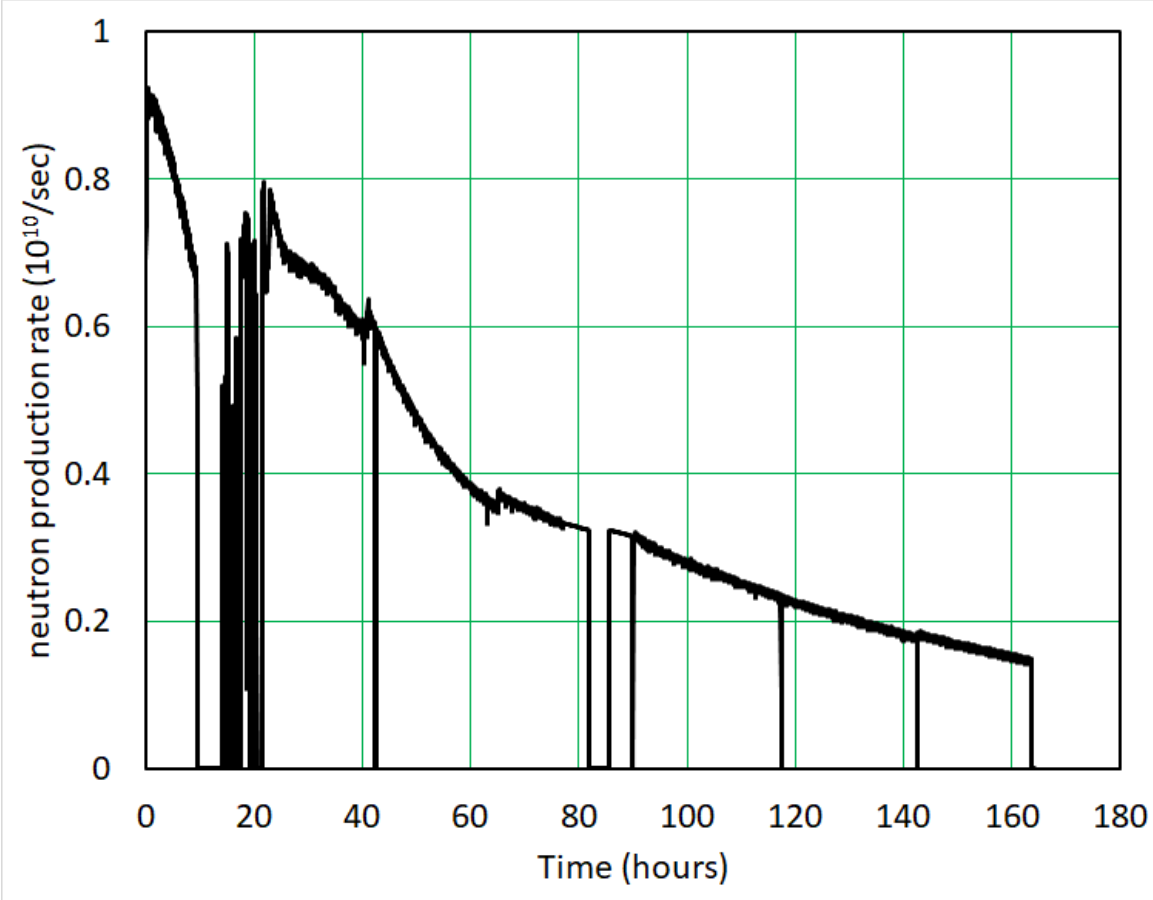

Figure 25. Neutron Test - Neutron Production Rate

5.2.2.2 Carbon Ion Testing

Section 5.2.1.2 previously discussed the test approach for carbon ion testing. The first carbon ion test had a fluence of 9.693×10^7. No errors were detected for data sets 0 through 6 corresponding to grid locations 0 through 6. Errors were detected for data sets 7 through 9. No further data was recovered after set 9. The checkerboard error counts recorded during the first carbon test are plotted in Figure 26. All of the data was collected synchronously by Matlab, so the four possible values of the data were identical, unlike those shown in Figure 23. Error counts of -10 are used to indicate data was not transmitted or unrecoverable. The remaining points indicate individual error counts.

It appears that the sudden change in the plot near data point number 8×10^4 is when the beam shot was fired during collection of data set 8. It also appears that the beam shot in data set 9 was fired around data point number 12×10^4. A trend in error count over time could not be established because the error count changed erratically throughout the test. After data set 9, no further data was recovered and the current draw jumped to 0.6A. After re-pressurizing and opening the chamber, the DE-10 was observed to have two LEDs lit and the remaining LEDs off when all LEDs should have been in a dim "half-on" state. Some segments of the six 7-segment displays were also lit and none of the segments should have been lit. This indicates that a significant number of configuration errors occurred because electrical pathways to the LEDs were connected to force them on or off and other electrical pathways were connected to turn on the 7-segment displays. Resetting and reprogramming the DE-10 caused the LEDs and 7-segment displays to return to their expected appearance. The DE-10's current draw also returned to nominal levels and the DE-10 passed functional tests.

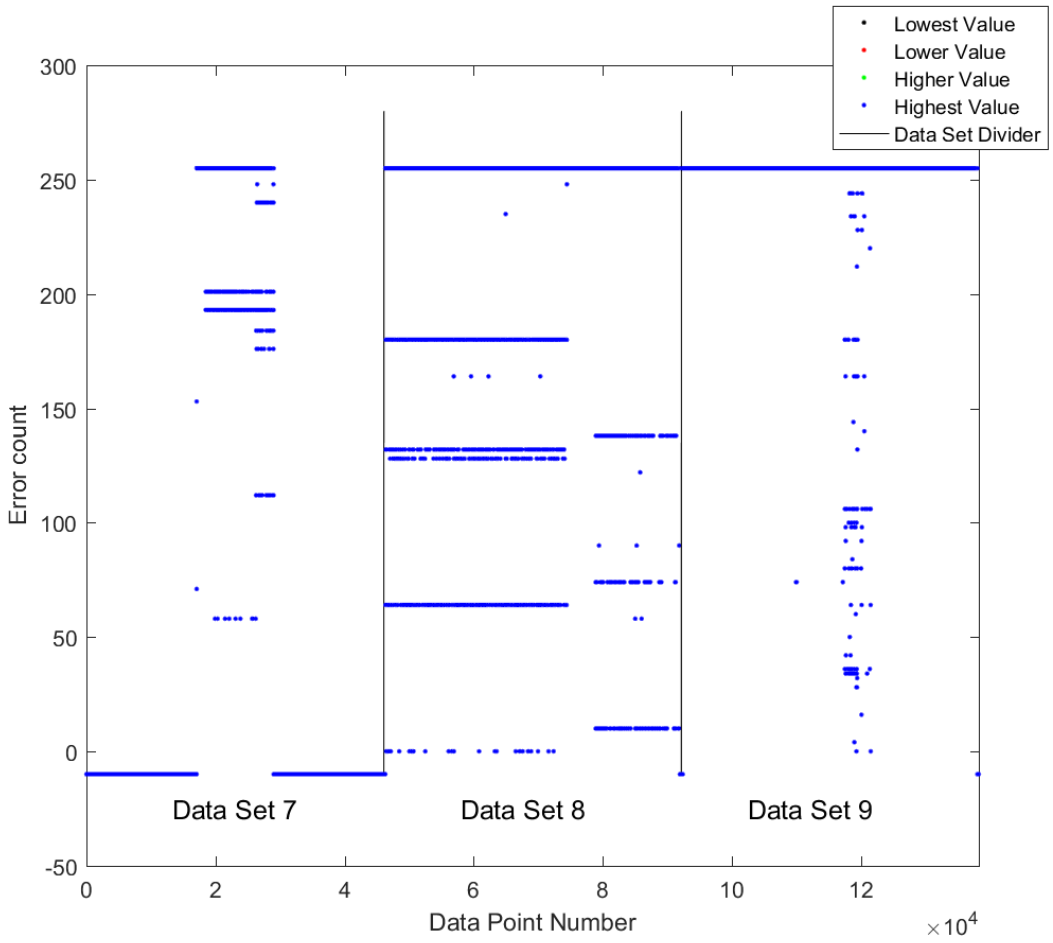

Figure 26. Carbon Test One

Data collected from the second carbon ion test with a fluence of 1.097×10^7 are plotted in Figure 27. No data was collected for sets 0 through 3, indicating that no errors occurred prior to set 4. There were severe synchronization issues for sets 4 through 7 that appear to have resolved during collection of set 7. Set 7 experienced synchronization issues such that the each recovered data point could have taken one of four possible values. Judging by the sets after set 7, it is assumed that error count values in red are the correct ones. The graph exhibits a general upward trend in error count on each subsequent test after test 3, though the shape is difficult to discern due to the lack of data from sets 4, 5, and 6. Sets 8 and 10 show evidence of a

major shift in the error count when the beam shot was fired. The rapidly changing error count at the end of set 11 may also indicate when the beam shot was fired, but data transmission stopped immediately afterward. The observed current draw after set 11 was 0.56A and no bytes were transmitted after set 11. The chamber was re-pressurized and opened in order to reset and reprogram the DE-10. The DE-10 then passed functional tests before proceeding to test three.

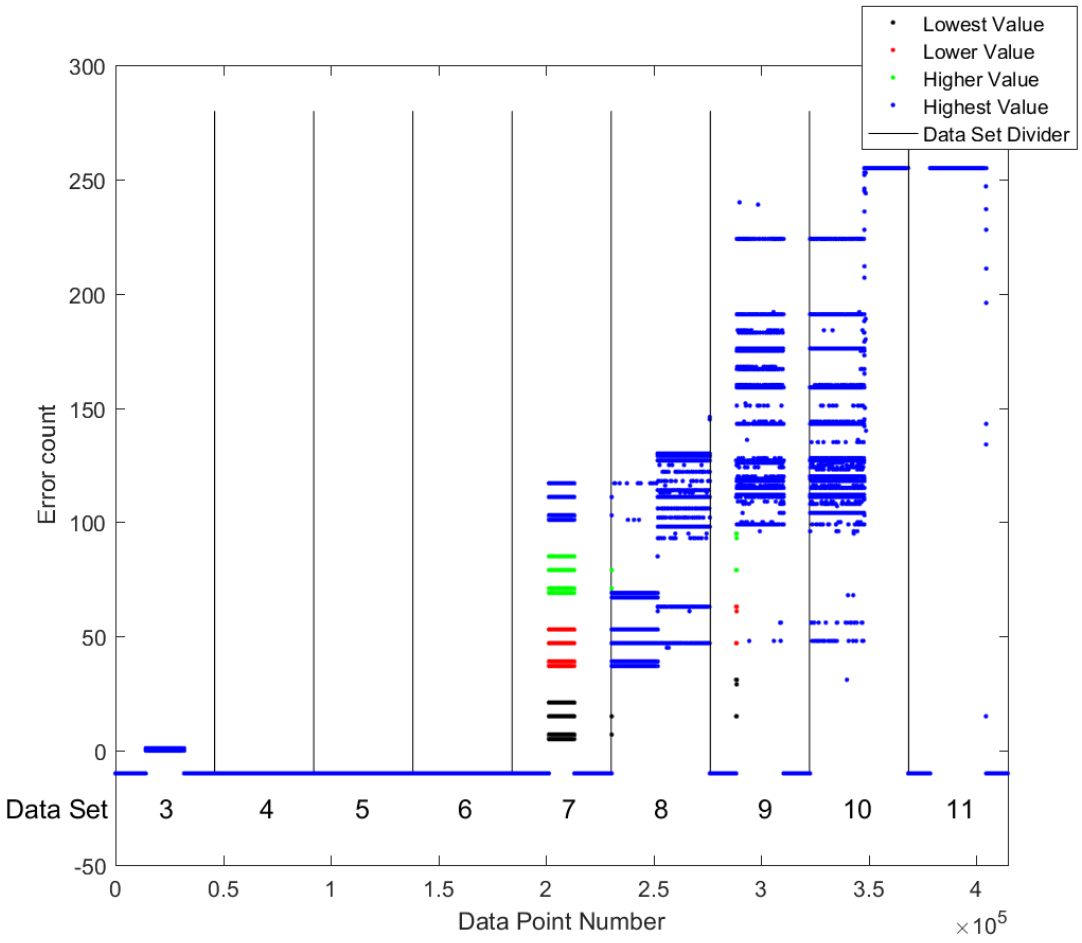

Figure 27. Carbon Test Two

Data collected from the third carbon ion test with a fluence of 9.834×10^5 are plotted in Figure 28. Due to the sparse nature of the data in sets 9, 10, and 11, no trend for error count over time could be established. Sets 7 and 8 do show an

increasing error count over time. The DE-10 stopped transmitting data after set 11. After re-pressurizing the chamber and resetting and reprogramming the DE-10, it passed functional tests.

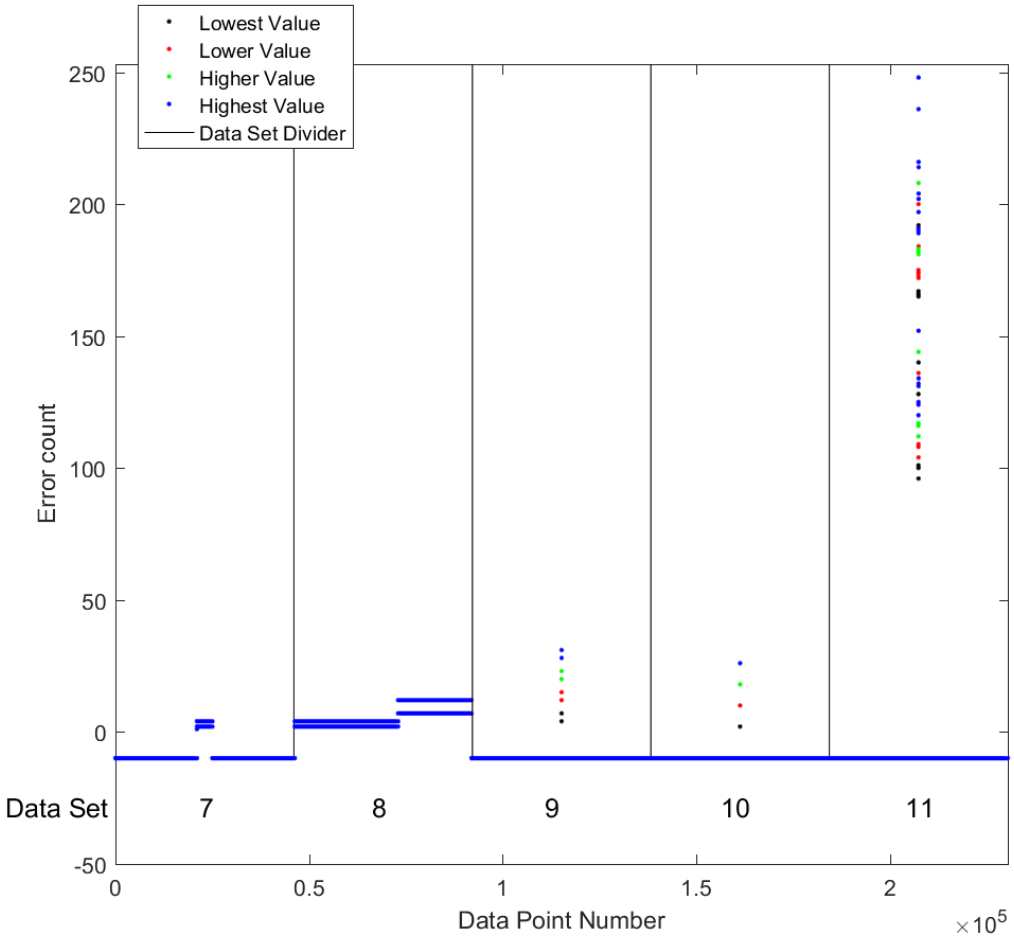

Figure 28. Carbon Test Three

The erratic nature of carbon ion experimental results made it impossible to determine a Cyclone V FIT rate when exposed to carbon ions. Future tests will need to be designed to determine the carbon ion radiation FIT rate.

5.2.2.3 Proton Testing

During the first proton test, the error count alternated between 0 and 1 indicating a possible configuration error. It is possible that a configuration error could replace one register input to an XNOR gate with a constant in one of the checkerboard locations. The other register in that checkerboard location would still alternate on every clock cycle causing the XNOR gate output and the error count to alternate.

During the second proton test, only 319 data points were recoverable due to synchronization problems. The third proton test yielded no recoverable data. This highlights the experiment limitations mentioned in Section 5.2.1.3 and the difficulty in acquiring data from radiation experiments. No FIT rate could be established for the Cyclone V when exposed to proton ion radiation. By the end of the second and third tests, the DE-10 stopped transmitting data. The DE-10 was drawing 0.6A and 0.9A at the end of the second and third tests respectively. After re-pressurizing the chamber, resetting, and reprogramming, the DE-10 passed functional tests after the second and third tests.

5.3 Error Injection Architecture

Now that the error rate has been determined to be 1 SEU/day, the next step is to determine how to inject errors. It was determined that the most effective method of injecting errors was to forcibly flip bits stored in registers in the Basic MIPS processor(s) of TMR, TSR, and AHR MIPS. These types of injected errors have the same impact as a SEU. These injected errors are not too dissimilar from SETs since SETs only become errors if they are latched into a register. Now that the mode of error injection is selected, the Basic MIPS registers to target must be determined. The Basic MIPS registers are a 32-bit instruction register and a 4-bit state machine register in the Controller as well as a 30-bit program counter register, thirty-one 32-bit

general purpose registers used by the program to compute results, and two registers used to delay signals necessary for branch instruction processing in the Datapath.

In order to ensure a meaningful, apples-to-apples timing and energy performance comparison between TMR, TSR, and AHR MIPS, it is necessary to target registers that will allow all three architectures to detect and correct the error. If an error is injected such that one architecture does not detect the error but another does, the architecture which fails to detect the error would not suffer any performance degradation, but would produce erroneous results. The next few paragraphs explore what could happen if an error were injected into each type of register in Basic MIPS and its effect on TMR and TSR MIPS. The effect on AHR MIPS is not explored in this discussion because AHR MIPS operates in either TMR or TSR mode at any given time and the effects would be identical to the effects on TMR and TSR MIPS.

Should an error occur in the instruction register such that an instruction is changed into an unrecognized instruction, Basic MIPS is designed to reject the instruction and request the instruction from memory once again. This would protect both TMR and TSR MIPS. In the event that an error causes one valid instruction to be changed to another valid instruction, TMR and TSR MIPS would both be able to detect an erroneous result in most cases. In contrast, if a TSR branch comparison instruction were changed to another instruction, TSR would realize an error it would not detect if one of the two copies of the value to be compared were also in error. TSR would also suffer from errors in the store word instructions if the instruction were corrupted such that it was no longer a store word instruction or if the instructions register reference were corrupted such that the wrong register value was stored to memory. TMR MIPS would detect all of these errors because the voter would detect that one of the three Basic MIPS processors was in error. An instruction register error does not lend itself to an apples-to-apples performance comparison between TMR and TSR MIPS.

If an error were to occur in the state machine register of Basic MIPS, TMR MIPS would be able to detect that one of its three Basic MIPS processors is out of step with the other two. In contrast, TSR MIPS only has one Basic MIPS processor and a state machine register error could cause TSR MIPS to jump to an unrecoverable state or skip over important state transitions when conducting branch comparisons or store word instructions which would go undetected. State machine register errors are also not a good way to compare TMR and TSR MIPS performance in the presence of errors.

A program counter error would be detected by TMR MIPS because one of the three Basic MIPS processors would attempt to read an instruction from a different memory address than the other two. A program counter error in TSR MIPS could cause a significant portion of the TSR MIPS program to be skipped. It could also cause a comparison instruction or a store word instruction to be skipped and allow an undetected error to occur. Therefore, program counter errors do not provide an apples-to-apples comparison between TMR and TSR MIPS.

An error injected to any one of the general purpose registers would be detected and corrected by both TMR and TSR MIPS. TMR MIPS would detect the error when the data that one Basic MIPS processor is attempting to write to memory differs from the other two. TSR MIPS would detect the error when the branch comparison instruction is evaluated and determines that the duplicated registers are not equal. This represents an apples-to-apples comparison between TMR and TSR MIPS.

An error injected into the delay registers used by branch instructions could cause a branch to be not taken when it should be taken or a branch to be taken when it should not be taken. TMR MIPS would detect this error so long as only one Basic MIPS processor is affected by this error. TSR MIPS would not detect this error and there is a potential that an error in one of two duplicated registers would be missed

by the branch comparison instruction should a branch delay register error occur. This is also a case where an error would be detected by TMR MIPS and not TSR MIPS and is therefore not a good way to compare TMR and TSR MIPS performance.

Based on this thought experiment, errors will only be injected into general purpose registers. The next step is to determine when to inject the errors. The determination was made to only inject errors into registers that are going to be stored to memory and to do so immediately before they are stored in memory. The errors are injected at the beginning of the clock cycle in which an instruction to store a register value to memory is being read from memory for TMR MIPS. For TSR MIPS, the errors are injected at the beginning of the clock cycle in which the branch instruction before a store word instruction is being read from memory. This ensures that both TMR and TSR MIPS are able to detect and correct the error. If other registers were selected that serve as intermediate results for later instructions that eventually store another register to memory, the error would propagate through multiple instructions before being detected. This method would also produce detectable errors, but would make it more difficult to predict exactly when an error would be detected for simulation and timing analysis purposes.

Error injection is physically implemented by adding hardware to the Basic MIPS architecture. The hardware allows a bit flip to be injected to a specific register at a specific time. That specific time is when the Basic MIPS Controller FSM is in state 0, the program counter is at a store word instruction in a TMR MIPS program and a branch comparison instruction in TSR MIPS, and the loop counter in R31 is also at a specific value. The hardware then flips a bit in the user definable register which is about to be stored to memory. This is accomplished by overriding the register select signal to the GPR Bank to write to the pre-selected register. The value written to the pre-selected register is the output of that register with a pre-selected bit from that

register inverted. A schematic showing the error injection module integrated into the Basic MIPS Datapath is shown in Figure 29. The original Datapath is shown in Figure 30 and can also be located in the "Triple Modular Redundancy MIPS Architecture Version 1.4" report [38]. Changes to the original Datapath to add error injection are shown in red in Figure 29.

Note that the **Error Inject** module's data input is connected to **GPR Bank** register output *o_Q##* to indicate that the register number can be any number between 1 and 31. The *i_state* input comes from the Basic MIPS **Controller**. The *i_PC* input is from the output of the PC Register. The **Datapath** is modified to include a multiplexer between the original multiplexer to select the **GPR Bank** *i_data* input and the updated *i_data* input. The **Datapath** is also modified to include a multiplexer between the original multiplexer to select the **GPR Bank** *i_sel* input and the *i_sel* input. These multiplexers normally allow the original signals to pass through to the **GPR Bank**, but switch to the input defined by the **Error Inject** module when an error is to be injected. This is accomplished by the **Error Inject** module changing the output of *o_error* to 1 when an error is to be injected and leaving *o_error* at 0 when no error is to be injected.

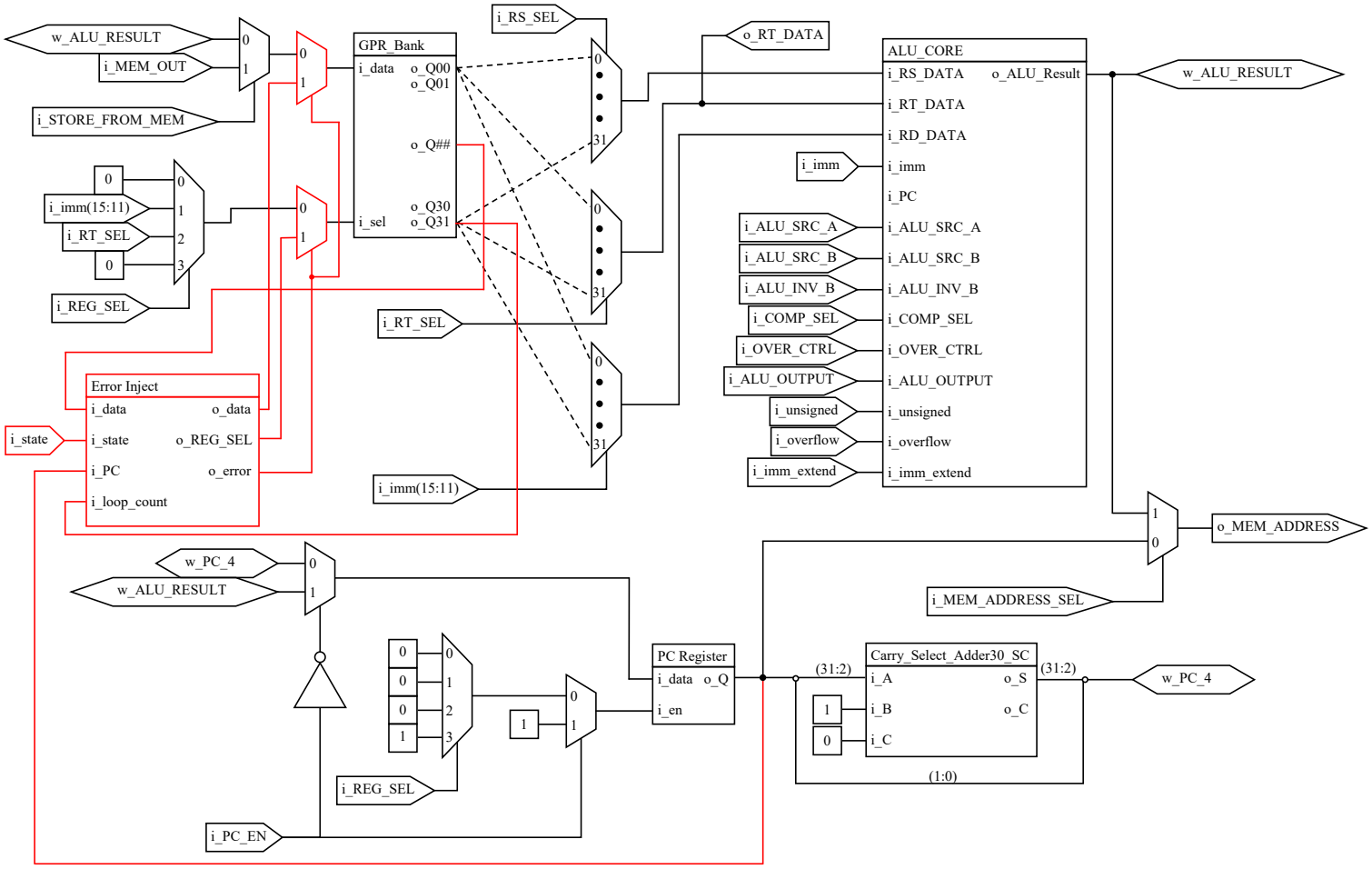

Figure 29. Basic MIPS Datapath with Error Injection Schematic

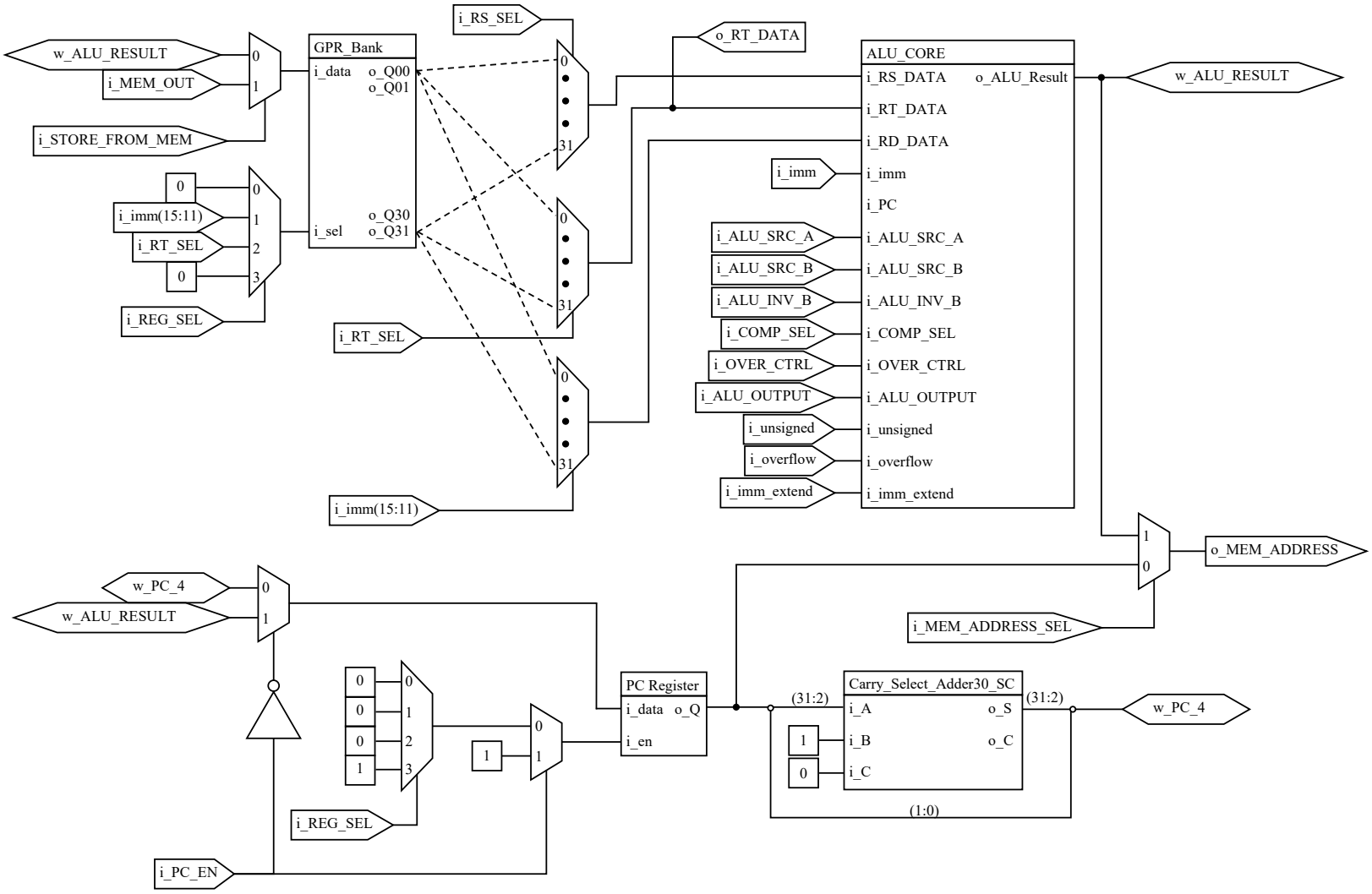

Figure 30. Basic MIPS Datapath Schematic

5.4 Software Simulation with Error Injection

Software simulation with error injection will be used to determine the effects of error injection on the various architectures in terms of time and energy to complete programs. The runtimes of equivalent programs across the various architectures can be compared when looking at best-case and worst-case errors. The number of errors to inject into a program is one error per program iteration. This was determined from experiments performed to assess the vulnerability of the Cyclone V FPGA as described in Sections 5.2.1 and 5.2.2. In neutron testing, the Cyclone V FPGA experienced an average of 1 error every 360 seconds (and an adjusted rate of 1 error every 3600 seconds. See Section 5.2.2.1). Since each program is expected to take about 50 milliseconds or less to run based upon the error free software simulation results in Section 4.4, one error per program run represents the maximum number of errors that would ever be expected to occur during any one program run.

5.4.1 Runtime Calculations

These simulations are able to provide timing information on error recovery operations for TMR MIPS, TSR MIPS, and AHR MIPS that could not previously be determined during error free operation. The simulations also establish upper and lower bounds for program runtime based on the best-case and worst-case scenarios for all programs.

These simulations cannot be run to completion due to the resource constraints of the computer running the simulations. Instead, the save/restore point and error injection times were modified to capture the time required to recover from an error. This timing information was also used to determine the amount of time from the start of save/restore point creation to encountering an error at the end of save/restore point creation for TMR and TSR MIPS worst-case scenarios which will be discussed shortly.

The timing information is used to calculate the total program runtime.

5.4.1.1 TMR MIPS Runtime

The first error scenario for TMR MIPS is a single processor error referred to as a Type A error. In this scenario, the program resumes from the same point at which the error occurred after error recovery is completed. The Type A error is illustrated in Figure 31. This figure shows a program executing from start to end in TMR MIPS. The first instruction is executed at the start and the last instruction is executed at the end. Save/restore points are created where indicated by "SRP". The red "X" indicates that an error occurred in Basic MIPS processor 1 while the green circles indicate that Basic MIPS processors 0 and 2 were in agreement when the error was detected. The blue arrows indicate that error recovery returned code execution to the point at which the error occurred.

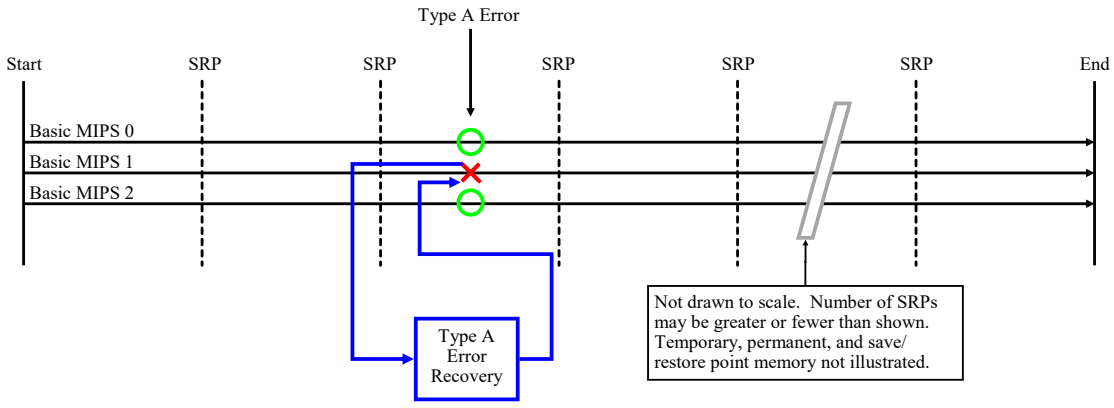

Figure 31. TMR MIPS Type A Error Timing Diagram

The runtime for a TMR MIPS Type A error is given in Equation 33 where $T_{TMR\ MIPS}$ is the time for TMR MIPS to complete a program in the absence of an error given in Equation 3, $T_{TMR\ ttdA}$ is the time it takes TMR MIPS to detect the error, $T_{TMR\ recA}$ is the time to recover from a single processor error, $T_{TMR\ retA}$ is the time to return to the instruction at which the error occurred, and $T_{TMR\ repA}$ is the

time required to repeat the instruction at which the error occurred. The last four of these values are determined from the simulation.

$$T_{TMR\ ErrA} = T_{TMR\ MIPS} + T_{TMR\ ttdA} + T_{TMR\ recA} + T_{TMR\ retA} + T_{TMR\ repA} \quad (33)$$

The second error scenario for TMR MIPS is a multiple processor error (possibly caused by a multiple-bit upset) referred to as a Type B error. In this scenario, the TMR Voter cannot determine if any processor is correct, so it resets all three processors and reloads their internal states from the most recently created save/restore point. This is illustrated in Figure 32 where all three processors are marked with a red "X" to indicate that the voter cannot determine which of the processors are correct. The blue arrows indicate that error recovery returned code execution to the most recent save/restore point.

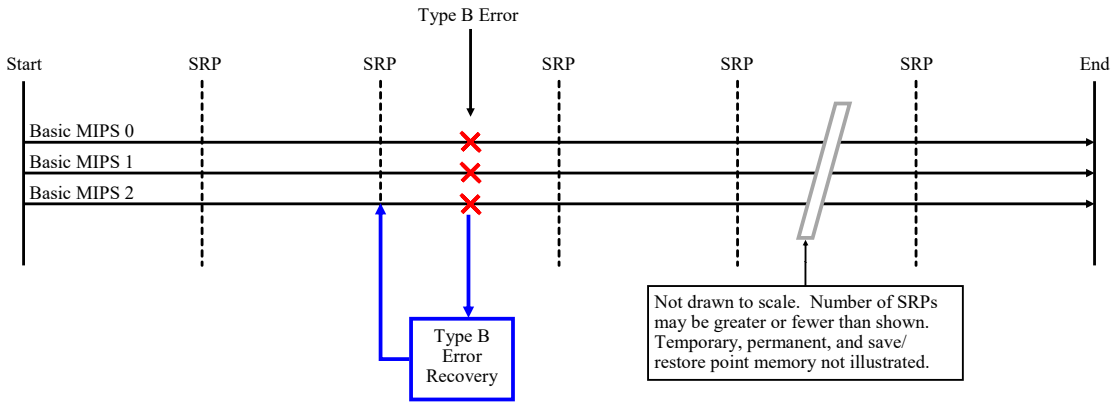

Figure 32. TMR MIPS Type B Error Timing Diagram

The Type B error is bounded by best-case and worst-case error scenarios. The best-case error minimizes the number of instructions which must be executed after error recovery to return to the point in the program at which the error occurred while the worst-case error maximizes this same number of instructions.

The Type B Best-case error occurs immediately after creation of a save/restore point. The error is injected immediately prior to the first store word instruction after creating a save/restore point. The error is injected into the register to be stored to memory. Specifically, two different bits of this register are inverted on two different processors. For example, R1 on one processor might be changed from a 0 to 32 while R1 on another processor might be changed from a 0 to 128. The Type B Best-case error is shown in Figure 33.

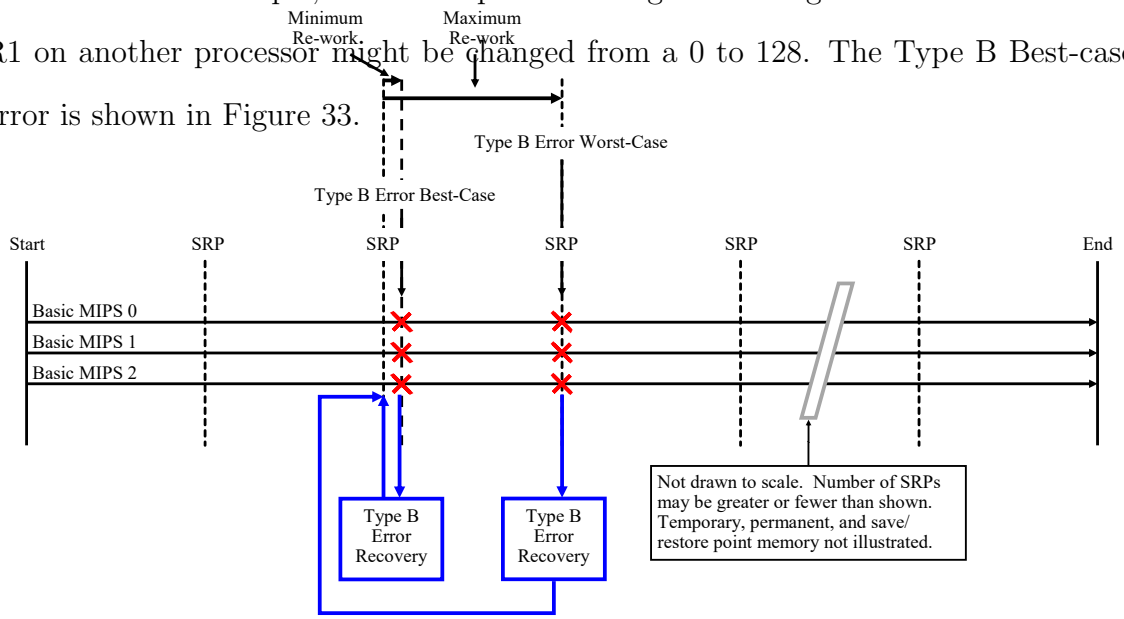

Figure 33. TMR MIPS Type B Best- and Worst-Case Error Timing Diagram

The runtime for TMR MIPS Type B Best-case error is given in Equation 34 where $T_{TMR\ ttdB}$ is the time it takes TMR MIPS to detect the error, $T_{TMR\ recB}$ is the time to recover from a multiple processor error, $T_{TMR\ retB\ Best}$ is the time to re-accomplish the instructions between the last completed save/restore point and the instruction at which the error occurred. The time to detect the error and the time to recover from the error are determined from the simulation, but the time to return from the error to the point at which the error occurred is determined by analysis.

$$T_{TMR\ ErrB\ Best} = T_{TMR\ MIPS} + T_{TMR\ ttdB} + T_{TMR\ recB} + T_{TMR\ retB\ Best} \quad (34)$$

While there are many TMR Type B errors, there can only be one Best-case error. There are many nearly Best-case errors, one after each save/restore point, but the absolute Best-case error is the one that minimizes the number of instructions between the return from save/restore point creation and the store word instruction following it. In order to determine which pairing of save/restore point creation and store word instructions has the shortest distance between them, the loop count and instruction index of every save/restore point creation and store word instruction must be determined. The store word instruction indices are simply located by examining the program. Equation 35 shows how to calculate where save/restore point creation occurs where SI_{TMR} is a vector containing the instruction index in the TMR program where save/restore points are created, SL_{TMR} is a vector containing the program loop count values where the save/restore points are created, and ST_{TMR} is a vector containing the amount of time from the beginning of the program to the points at which save/restore points are created. ST_{TMR} is not used now in calculating the TMR Type B-Best program completion time, but will be used shortly.

$$
\begin{aligned}
&for\ m = 0\ to\ n_{SRP} - 1 \\
&\quad if\ m = 0 \\
&\qquad SI_{TMR}(m+1) = 1 \\
&\qquad SL_{TMR}(m+1) = 0 \\
&\qquad ST_{TMR}(m+1) = 0 \\
&\quad else \\
&\qquad SI_{TMR}(m+1) = mod(m \cdot n_{save} - n_{TMR\ init}, N_{TMR}) + n_{TMR\_init} + 1 \\
&\qquad SL_{TMR}(m+1) = \left\lfloor \frac{m \cdot n_{save} - n_{TMR\ init}}{n_{TMR}} \right\rfloor \\
&\qquad T_{add} = \textstyle\sum_{n=n_{TMR\ init}+1}^{SI_{TMR}(m+1)-1} t_{I_{TMR\ n}} \\
&\qquad ST_{TMR}(m+1) = T_{TMR\ init} + T_{TMR\ loop} \cdot SL_{TMR}(m+1) + T_{add} + \cdots \\
&\qquad (m-1) \cdot T_{TMR\ SRP} \\
&\quad end \\
&end
\end{aligned}
\tag{35}
$$

The next step is to compute all possible differences (SD_1) between save/restore point indices and store word indices as shown in Equation 36 where SI_{TMR}^T is the transpose of SI_{TMR}. This formula states SD_1 is a matrix of row vectors such that the n^{th} row subtracts each value of SI_{TMR} from the n^{th} SW_{TMR} value. Note that SW_{TMR} is a vector containing the indices of every store word instruction in a program.

$$
\begin{aligned}
&for\ n = 1\ to\ length(SW_{TMR}) \\
&\quad SD_1(n,:) = SW_{TMR}(n) - SI_{TMR}^T \\
&end
\end{aligned}
\tag{36}
$$

Next, because some of the values in SD_1 may be negative because a save/restore point may occur at the end of one loop and the next store word may occur at the

beginning of the next loop, SD_1 is modified so that all values are positive as shown in Equation 37. In this equation, the "<" and ">" operators are logical operators that populate a matrix with ones or zeros depending on whether the individual matrix entries are less than or greater than the argument to the right of the operator. The term $SD_1 . \cdot (SD_1 > 0)$ creates a matrix with all the positive values of SD_1 and where all the negative values of SD_1 are set to zero (the ".·" operator denotes elementwise multiplication). The term $SD_1 . \cdot (SD_1 < 0)$ creates a matrix with all the negative values of SD_1 and where all the positive values of SD_1 are set to zero. The term $N_{TMR} \cdot (SD_1 < 0)$ creates a matrix where all the negative values of SD_1 are replaced by N_{TMR} and all the positive values of SD_1 are set to zero. The term $SD_1 . \cdot (SD_1 < 0) + N_{TMR} \cdot (SD_1 < 0)$ creates a matrix where all the negative values of SD_1 are replaced by the positive number of instructions from the save/restore point at the end of a loop to the store word instruction at the beginning of the next loop and accounts for the fact that code execution jumped from the end of the loop back to the start of the loop. Finally, SD_2 contains all the positive instruction distances between save/restore points and the store words following them.

$$SD_2 = SD_1 . \cdot (SD_1 > 0) + (SD_1 . \cdot (SD_1 < 0) + N_{TMR} \cdot (SD_1 < 0)) \qquad (37)$$

The next step is to determine the minimum distance between a save/restore point creation and a store word instruction. Equation 38 is used to calculate the minimum distance where min is a function that returns the minimum value of each column vector of a matrix in the row vector a_1 and returns the index of each minimum value in each column vector in the row vector b_1. For a vector, min returns the minimum value in c_1 and the index of the minimum value in d_1. The value d_1 is as an index into the columns of SD_2 and tells which column contains the minimum distance between a save/restore point and a store word. The value $b_1(d_1)$ is an index into the rows of

SD_2 and tells which row contains the minimum distance between a save/restore point and a store word. Because the columns of SD_2 correspond to the save/restore point indices SI_{TMR} and the rows correspond to the store word indices SW_{TMR}, $SI_{TMR}(d_1)$ is the address of the instruction at which the save/restore point is created closest to the store word instruction at the address specified to $SW_{TMR}(b_1(d_1))$. In other words, this is the absolute shortest distance between the creation of a save/restore point and when an error could occur at a store word and constitutes the best-case multiple processor error for TMR MIPS.

$$\begin{aligned} [a_1, b_1] &= min(SD_2) \\ [c_1, d_1] &= min(a_1) \end{aligned} \tag{38}$$

The formula for determining $T_{TMR\ retB\ Best}$ is now presented in Equation 39. The reason for the if-else statement is because the program is in a loop and $SW_{TMR}(b_1(d_1))$ could be less than $SI_{TMR}(d_1)$.

$$\begin{aligned} &if\ SW_{TMR}(b_1(d_1)) \geq SI_{TMR}(d_1) \\ &\quad T_{TMR\ retB\ Best} = \textstyle\sum_{n=SI_{TMR}(d_1)}^{SW_{TMR}(b_1(d_1))} t_{I_{TMR\ n}} \\ &else \\ &\quad T_{TMR\ retB\ Best} = \textstyle\sum_{n=SI_{TMR}(d_1)}^{n_{TMR\_init}+N_{TMR}} t_{I_{TMR\ n}} + \sum_{n=n_{TMR\_init}+1}^{SW_{TMR}(b_1(d_1))} t_{I_{TMR\ n}} \\ &end \end{aligned} \tag{39}$$

The definition of the *min* function in Equation 38 presents an interesting situation when SW_{TMR} or SI_{TMR} is a scalar rather than a vector. In this situation, SD_2 will be a vector instead of a matrix and performing the operations in Equation 38 will not provide usable results for proper indexing into SW and SI in Equation 39. If SW_{TMR} is a scalar, b_1 is used as the indexing variable into SI_{TMR} and no index variable is

used for SW_{TMR} because it is a scalar. These adjustments are made to Equation 39 as shown in Equation 40. If SI_{TMR} is a scalar, b_1 is used as the indexing variable into SW_{TMR} and no index variable is used for SI_{TMR} because it is a scalar. These adjustments are made to Equation 39 as shown in Equation 41.

$$
\begin{aligned}
&if SW_{TMR} \geq SI_{TMR}(b_1) \\
&\quad T_{TMR\ retB\ Best2} = \textstyle\sum_{n=SI_{TMR}(b_1)}^{SW_{TMR}} t_{I_{TMR\ n}} \\
&else \\
&\quad T_{TMR\ retB\ Best2} = \textstyle\sum_{n=SI_{TMR}(b_1)}^{n_{TMR\_init}+N_{TMR}} t_{I_{TMR\ n}} + \sum_{n=n_{TMR\_init}+1}^{SW_{TMR}} t_{I_{TMR\ n}} \\
&end
\end{aligned}
\tag{40}
$$

$$
\begin{aligned}
&if SW_{TMR}(b_1) \geq SI_{TMR} \\
&\quad T_{TMR\ retB\ Best} = \textstyle\sum_{n=SI_{TMR}}^{SW_{TMR}(b_1)} t_{I_{TMR\ n}} \\
&else \\
&\quad T_{TMR\ retB\ Best} = \textstyle\sum_{n=SI_{TMR}}^{n_{TMR\_init}+N_{TMR}} t_{I_{TMR\ n}} + \sum_{n=n_{TMR\_init}+1}^{SW_{TMR}(b_1)} t_{I_{TMR\ n}} \\
&end
\end{aligned}
\tag{41}
$$

The Type B Worst-case error maximizes the number of instructions which must be executed after error recovery to return to the point in the program at which the error occurred. Therefore, the worst-case error occurs at the end of creating a save/restore point so that the error is detected before successful save/restore point creation. The multiple bit error is injected when attempting to write the loop counter when creating the save/restore point such that a multiple processor error is detected and triggers recovery operations. In this scenario, 10,000 instructions and save/restore

point creation must be repeated to return to the point in the program at which the error occurred. The Type B Worst-case error is shown in Figure 33.

The runtime for TMR MIPS Type B Worst-case error is given in Equation 42 where $T_{TMR\ SRP\ Err}$ is the time it takes TMR MIPS to encounter an error during creation of a save/restore point when the error occurs in the program counter of multiple processors when attempting to save the program counter to memory, $T_{TMR\ recB}$ is the time to recover from a multiple processor error, $T_{TMR\ retB\ Worst}$ is the time to return to the instruction at which the error occurred. The time $T_{TMR\ SRP\ Err}$ is determined from the simulation, but $T_{TMR\ retB\ Worst}$ is determined by analysis.

$$T_{TMR\ ErrB\ Worst} = T_{TMR\ MIPS} + T_{TMR\ SRP\ Err} + \cdots \\ T_{TMR\ recB} + T_{TMR\ retB\ Worst} \tag{42}$$

To compute the worst-case scenario time to return to the instruction at which the error occurred, the worst-case time between save points must first be determined according to Equation 43 where ST_{TMR} was previously defined in Equation 35, $SL_{TMR}(m)$ is the number of full loops completed by the time the m^{th} save/restore point creation is reached, T_{add} is the time from the start of the loop in which the save/restore point is created to the instruction in that loop at which the save/restore point is created, $ST_{TMR}(m)$ is the time from the beginning of the program to the time at which the m^{th} save/restore point creation begins, SDT_{TMR} is the save time difference between consecutive save points, and $W_{SI_{TMR}}$ is the index of the worst-case SDT_{TMR}. The value of SDT_{TMR} is obtained by subtracting the 1st value of ST_{TMR} from the second value, the second value from the third, and so on until the $(n_{SRP} - 1)^{th}$ value is subtracted from the n_{SRP}^{th}. The maximum value of SDT_{TMR} is $T_{TMR\ retB\ Worst}$.

$$SDT_{TMR} = ST_{TMR} - [0, ST_{TMR}(1\ to\ length(ST_{TMR}) - 1)]$$
$$[T_{TMR\ retB\ Worst}, W_{SI_{TMR}}] = max(SDT_{TMR}) \quad (43)$$

While multiple-bit upsets are assumed to be unlikely events, they are included in order to test TMR MIPS Type B Best-case and Worst-case scenarios that would not otherwise occur if only SEUs and SETs were permitted. The TMR MIPS multiple processor error upset errors are referred to as Type B errors and are further denoted as Type B-Best and Type B-Worst for the best-case and worst-case multiple processor errors respectively. It should be expected that Type A and Type B-Best errors would minimize the time and energy added to complete a program while Type B-Worst errors would maximize the time and energy added to complete a program.

5.4.1.2 TSR MIPS Runtime

TSR MIPS error recovery process is much like the TMR MIPS Type B error recovery process in that TSR MIPS always returns to the most recent save/restore point. This is illustrated in Figure 34 which also shows how the program jumps from an error, to error recovery code, then back to the save/restore point. TSR MIPS errors are similarly divided into best-case and worst-case scenarios.

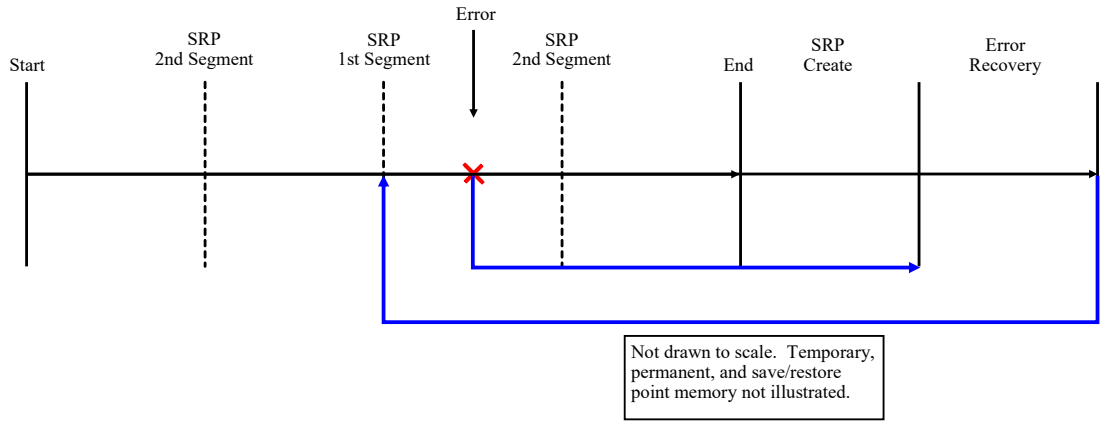

Figure 34. TSR MIPS Error Timing Diagram

The best-case scenario minimizes the number of instructions which must be executed after error recovery to return to the point in the program at which the error occurred. Therefore, the best-case error occurs immediately after creation of a save/restore point. The error is injected immediately prior to the branch comparison instruction before the first store word instruction after creating a save/restore point. The error is injected into one of the registers to be compared. The TSR MIPS Best-case error is shown in Figure 35.

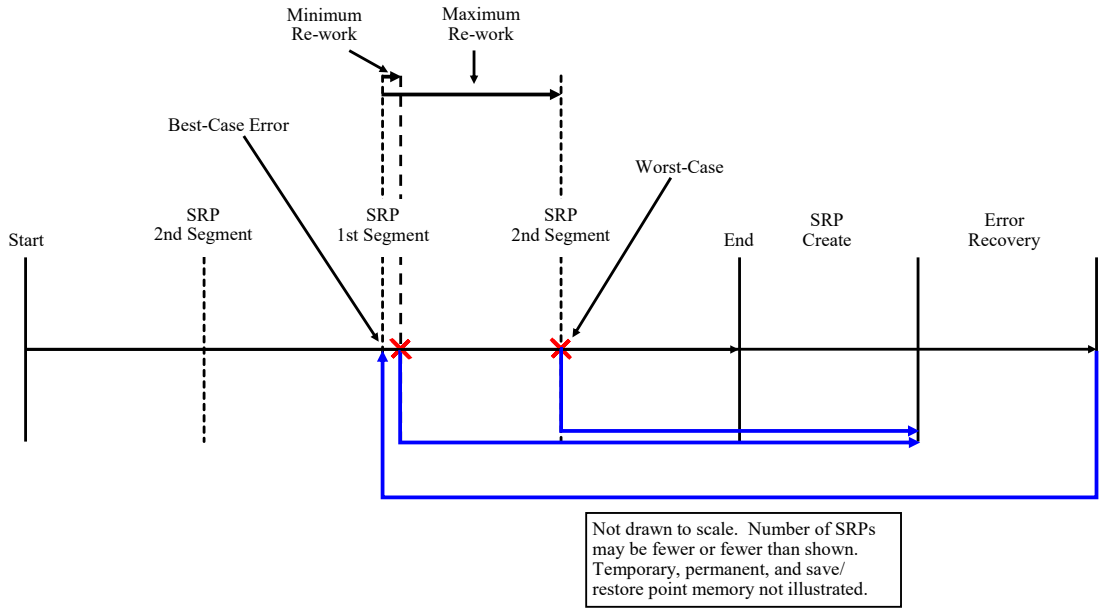

Figure 35. TSR MIPS Best- and Worst-Case Error Timing Diagram

The TSR MIPS Best-case error is computed using Equation 44 where $T_{TSR\ MIPS}$ was previously defined in Equation 6, $T_{TSR\ Rec}$ is the time to perform error recovery operations and is determined from simulation results, and $T_{TSR\ Ret}$ is the time needed to return from the most recent save/restore point to the instruction at which the error occurred.

$$T_{TSR\ Best} = T_{TSR\ MIPS} + T_{TSR\ Rec} + T_{TSR\ Ret} \tag{44}$$

The time to return to the instruction at which the error occurred is determined using Equation 45 where $n_{TSR\ init}$ is the number of instructions needed to initialize a TSR program (4 instructions) and SW_{TSR} is a vector containing the instruction indices of all store word instructions in a TSR MIPS program.

$$T_{TSR\ ret} = \sum_{n=N_{TSR}-3}^{N_{TSR}} t_{I_{TSR\ n}} + \sum_{n=n_{TSR\ init}+1}^{SW_{TSR}(1)} t_{I_{TSR\ n}} \tag{45}$$

The TSR MIPS Worst-case scenario maximizes the number of instructions which must be executed after error recovery to return to the point in the program at which the error occurred. Therefore, the worst-case error occurs at the end of creating a save/restore point. This error would specifically target the loop counter, which is the last register to be written to the save/restore point during save/restore point creation. This error would force TSR MIPS to restore itself from the previous save/restore point and then proceed past the end of the next save/restore point creation, which means completing 250 program loops all over again. Additionally, the worst-case error will occur when creating the save/restore point in the second segment of save/restore point memory rather than the first segment because the second segment takes longer to create. The TSR MIPS Worst-case error is also shown in Figure 35.

The TSR MIPS Worst-case error is computed using Equation 46 where $T_{TSR\ loop}$ was previously defined in Equation 7 and $T_{TSR\ SRP1\ Err}$ is the time from the start of save/restore point creation to the time at which an error is detected in the difference between the loop counter and duplicate loop counter. The value of $T_{TSR\ SRP1\ Err}$ is determined from the simulation. The term $\sum_{n=N_{TSR}-3}^{N_{TSR}} t_{I_{TSR\ n}}$ is the time to complete the loop after performing error recovery and the term $250 \cdot T_{TSR\ loop}$ is the time to re-complete the 250 loops between save/restore points.

$$T_{TSR\ Worst} = T_{TSR\ MIPS} + T_{TSR\ Rec} + \sum_{n=N_{TSR}-3}^{N_{TSR}} t_{I_{TSR\ n}} + \cdots$$
$$250 \cdot T_{TSR\ loop} + T_{TSR\ SRP1\ Err} \tag{46}$$

5.4.1.3 AHR MIPS Runtime

AHR MIPS suffers from all the same errors as TMR MIPS when in TMR mode and TSR MIPS when in TSR mode; however, the calculation of time to complete a program is complicated by the TMR to TSR transition.

TMR errors can affect the location of the transition because the transition only occurs after 15,000 instructions are completed without an error. This means that 15,000 instructions must be completed after the initial error before the transition can take place. This means a greater portion of the program will complete in TMR and a smaller portion in TSR than if no error had occurred. This becomes more evident as the error types are examined in detail.

When AHR MIPS encounters a TMR Type A error, it handles the error the same way that TMR MIPS would. However, the location of the error relative to the TMR to TSR transition point will impact the program's runtime. If the error occurs early in the program, such as at the first store word instruction in the program, the TMR to TSR transition point is only moved by a few instructions as shown in Figure 36. This is referred to as a Type A Early error and it has a minimal impact on the program runtime.

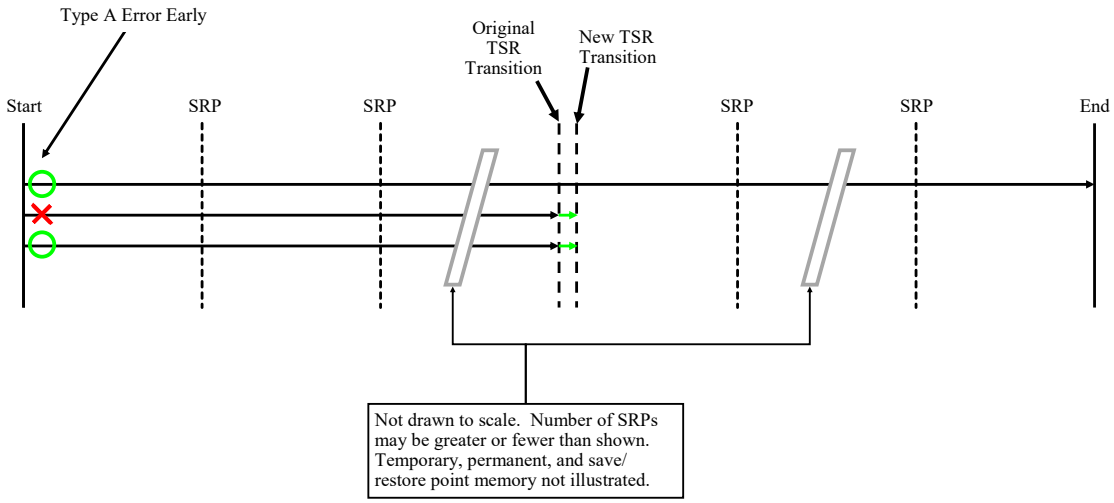

Figure 36. AHR MIPS TMR Type A Early Error Timing Diagram

Equation 47 shows how to calculate the AHR MIPS Type A Early error timing where $P_{loops\ TMR\ A\ Early}$ is the new transition point loop count determined according to Equation 48 and $n_{CSRP\ A\ Early}$ is the number of save/restore points to create prior to the transition determined by Equation 49. The TMR to TSR transition point determines how many save/restore points are created in TMR and TSR mode. The TMR mode save/restore points are determined by $n_{CSRP\ A\ Early}$, but the number of TSR mode save/restore points depends on where the transition occurs relative to the creation point for the TSR mode save/restore points which only occur at 250, 500, and 750 loops. This is the rationale for the if-else statements in these equations. There is also a possibility that the Type A error may push the TMR to TSR transition point out past the end of the program, in which case, AHR MIPS never enters TSR mode. Also note that $t_{CTMRAE\ TMR}$ and $t_{CTMRAE\ TSR}$ are the time AHR MIPS spends in TMR and TSR mode respectively when encountering a TMR Type A Early error. The time spent in TMR and TSR are separated to make the energy calculations simpler.

$$
\begin{aligned}
&t_{nom\ AE} = t_{TMR\ init} + P_{loops\ TMR\ A\ Early} \cdot T_{TMR\ loop} + \cdots \\
&T_{TMR\ SRP} \cdot n_{CSRP\ A\ Early} \\
&t_{err\ AE} = T_{TMR\ ttdA} + T_{TMR\ recA} + T_{TMR\ retA} + T_{TMR\ repA} \\
&if\ P_{loops\ TMR\ A\ Early} < 250 \\
&\quad t_{CTMRAE\ TMR} = t_{nom\ AE} + t_{err\ AE} + t_{TMR \rightarrow TSR} \\
&\quad t_{CTMRAE\ TSR} = (n_{loops} - P_{loops\ TMR\ A\ Early}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + 2 \cdot T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip} \\
&elseif\ 250 \leq P_{loops\ TMR\ A\ Early} < 500 \\
&\quad t_{CTMRAE\ TMR} = t_{nom\ AE} + t_{err\ AE} + t_{TMR \rightarrow TSR} \\
&\quad t_{CTMRAE\ TSR} = (n_{loops} - P_{loops\ TMR\ A\ Early}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip} \\
&elseif\ 500 \leq P_{loops\ TMR\ A\ Early} < 750 \\
&\quad t_{CTMRAE\ TMR} = t_{nom\ AE} + t_{err\ AE} + t_{TMR \rightarrow TSR} \\
&\quad t_{CTMRAE\ TSR} = (n_{loops} - P_{loops\ TMR\ A\ Early}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP1} + T_{TSR\ conc} - \frac{2}{3} T_{TSR\ skip} \\
&elseif\ 750 \leq P_{loops\ TMR\ A\ Early} < n_{loops} \\
&\quad t_{CTMRAE\ TMR} = t_{nom\ AE} + t_{err\ AE} + t_{TMR \rightarrow TSR} \\
&\quad t_{CTMRAE\ TSR} = (n_{loops} - P_{loops\ TMR\ A\ Early}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ conc} \\
&elseif\ P_{loops\ TMR\ A\ Early} \geq n_{loops} \\
&\quad t_{CTMRAE\ TMR} = t_{TMR\ init} + n_{loops} \cdot T_{TMR\ loop} + \cdots \\
&\quad T_{TMR\ SRP} \cdot (n_{SRP} - 1) + t_{err\ AE} \\
&\quad t_{CTMRAE\ TSR} = 0 \\
&end \\
&T_{CTMR\ A\ Early} = t_{CTMRAE\ TMR} + t_{CTMRAE\ TSR}
\end{aligned}
\tag{47}
$$

$$P_{loops\ TMR\ A\ Early} = \left\lceil \frac{SW_{TMR}(1) + n_{transition} - n_{TMR\_init}}{N_{TMR}} \right\rceil \tag{48}$$

$$n_{CSRP\ A\ Early} = \left\lfloor \frac{P_{loops\ TMR\ A\ Early} \cdot N_{TMR} + n_{TMR\_init}}{n_{save}} \right\rfloor \tag{49}$$

If the TMR Type A error occurs late in the program, such as at the last store word instruction before the TMR to TSR transition, the TMR to TSR transition is moved by nearly 15,000 instructions past the point at which it would have occurred if there were no error. This is shown in Figure 37. This is referred to as a Type A Late error and it causes the program to execute more instructions in TMR MIPS and fewer instructions in TSR MIPS than if no error had occurred. The expected effect is a significantly shorter runtime and increased energy usage.

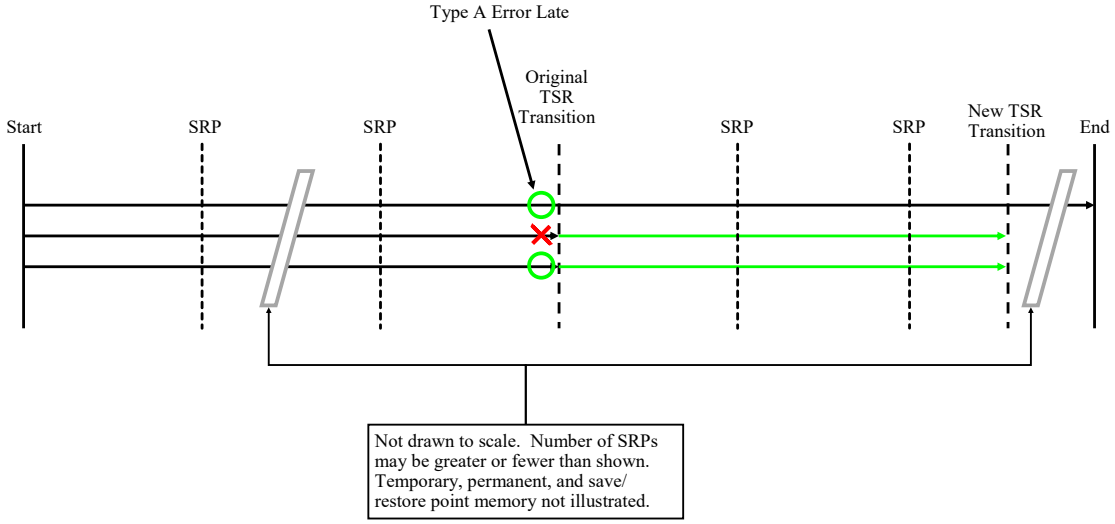

Figure 37. AHR MIPS TMR Type A Late Error Timing Diagram

Equation 50 shows how to calculate the AHR MIPS Type A Late error timing where $P_{loops\ TMR\ A\ Late}$ is the new transition point loop count determined according to Equation 51 and $n_{CSRP\ A\ Late}$ is the number of save/restore points to create prior to the transition determined by Equation 52. Also note that $t_{CTMRAL\ TMR}$ and $t_{CTMRAL\ TSR}$ are the time AHR MIPS spends in TMR and TSR mode respectively

when encountering a TMR Type A Late error.

$$
\begin{aligned}
&t_{nom\ AL} = t_{TMR\ init} + P_{loops\ TMR\ A\ Late} \cdot T_{TMR\ loop} + \cdots \\
&T_{TMR\ SRP} \cdot n_{CSRP\ A\ Late} \\
&t_{err\ AL} = T_{TMR\ ttdA} + T_{TMR\ recA} + T_{TMR\ retA} + T_{TMR\ repA} \\
&if\ P_{loops\ TMR\ A\ Late} < 250 \\
&\quad t_{CTMRAL\ TMR} = t_{nom\ AL} + t_{err\ AL} + t_{TMR \rightarrow TSR} \\
&\quad t_{CTMRAL\ TSR} = (n_{loops} - P_{loops\ TMR\ A\ Late}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + 2 \cdot T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip} \\
&elseif\ 250 \leq P_{loops\ TMR\ A\ Late} < 500 \\
&\quad t_{CTMRAL\ TMR} = t_{nom\ AL} + t_{err\ AL} + t_{TMR \rightarrow TSR} \\
&\quad t_{CTMRAL\ TSR} = (n_{loops} - P_{loops\ TMR\ A\ Late}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip} \\
&elseif\ 500 \leq P_{loops\ TMR\ A\ Late} < 750 \\
&\quad t_{CTMRAL\ TMR} = t_{nom\ AL} + t_{err\ AL} + t_{TMR \rightarrow TSR} \\
&\quad t_{CTMRAL\ TSR} = (n_{loops} - P_{loops\ TMR\ A\ Late}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP1} + T_{TSR\ conc} - \frac{2}{3} T_{TSR\ skip} \\
&elseif\ 750 \leq P_{loops\ TMR\ A\ Late} < n_{loops} \\
&\quad t_{CTMRAL\ TMR} = t_{nom\ AL} + t_{err\ AL} + t_{TMR \rightarrow TSR} \\
&\quad t_{CTMRAL\ TSR} = (n_{loops} - P_{loops\ TMR\ A\ Late}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ conc} \\
&elseif\ P_{loops\ TMR\ A\ Late} \geq n_{loops} \\
&\quad t_{CTMRAL\ TMR} = t_{TMR\ init} + n_{loops} \cdot T_{TMR\ loop} + \cdots \\
&\quad T_{TMR\ SRP} \cdot (n_{SRP} - 1) + t_{err\ AL} \\
&\quad t_{CTMRAL\ TSR} = 0 \\
&end \\
&T_{CTMR\ A\ Late} = t_{CTMRAL\ TMR} + t_{CTMRAL\ TSR}
\end{aligned}
\tag{50}
$$

$$P_{loops\ TMR\ A\ Late} = \left\lceil \frac{SW_{TMR}(length(SW_{TMR})) + n_{transition} - n_{TMR\_init}}{N_{TMR}} \right\rceil \quad (51)$$

$$n_{CSRP\ A\ Late} = \left\lfloor \frac{P_{loops\ TMR\ A\ Late} \cdot N_{TMR} + n_{TMR\_init}}{n_{save}} \right\rfloor \quad (52)$$

AHR MIPS may also encounter TMR MIPS Type B-Best errors early or late and these are referred to as TMR Type B-Best Early and TMR Type B-Best Late errors. As with the TMR Type A Early error, the TMR Type B-Best Early error has a minimal impact on runtime. As with the TMR Type A Late error, the TMR Type B-Best Late error is expected to significantly decrease runtime and increase energy usage. The TMR Type B-Best Early error is shown in Figure 38 while the TMR Type B-Best Late error is shown in Figure 39.

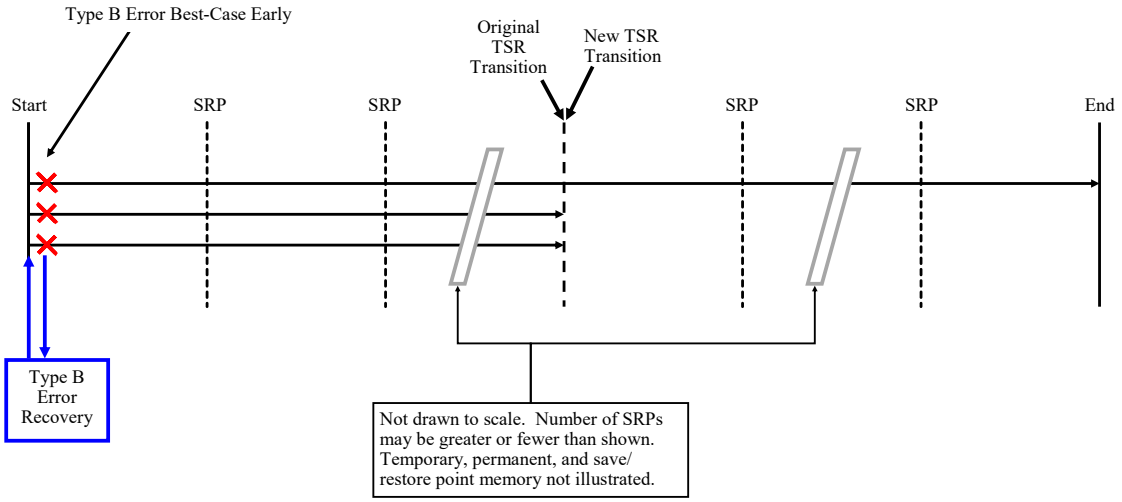

Figure 38. AHR MIPS TMR Type B Best-Case Early Error Timing Diagram

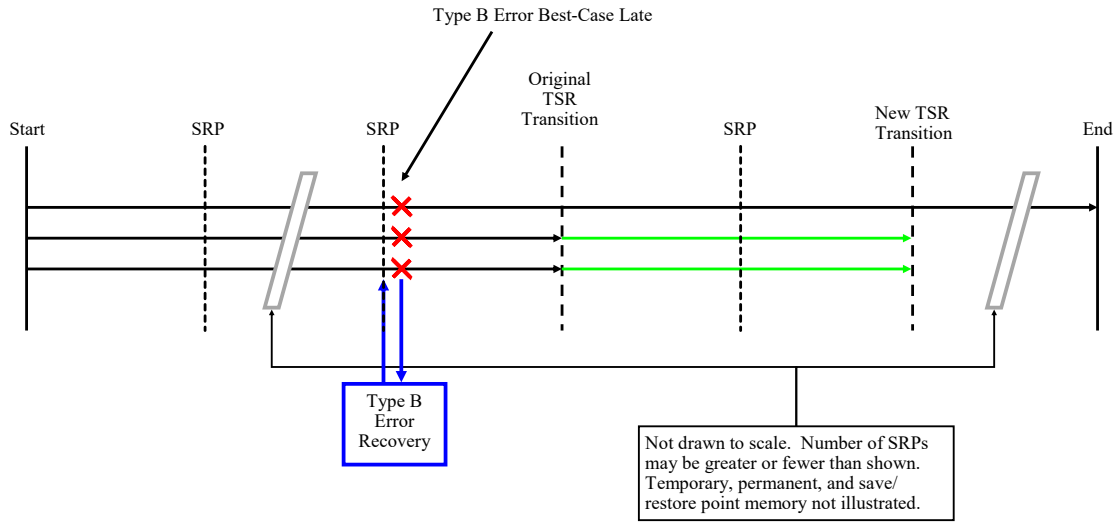

Figure 39. AHR MIPS TMR Type B Best-Case Late Error Timing Diagram

In order to determine the AHR MIPS runtime for programs with Type B-Best Early and Late errors, some of the variables used in computing TMR MIPS runtime for programs with Type B-Best errors need to be modified. The variable SI_{TMR} needs to be modified so it only contains instruction indices of save/restore points that occur before the original TMR to TSR transition was expected to take place. The values of SL_{TMR} and ST_{TMR} must also be updated. These updates are illustrated in Equation 53 where the $SL_{TMR}(SL_{TMR} < P_{loops})$ returns the vector of SL_{TMR} where the values of SL_{TMR} are less than the TMR to TSR transition point and all other values of the original SL_{TMR} vector are excluded.

$$\begin{aligned} SL_{CTMR} &= SL_{TMR}(SL_{TMR} < P_{loops}) \\ SI_{CTMR} &= SI_{TMR}(1 \; to \; length(SL_{TMR})) \\ ST_{CTMR} &= ST_{TMR}(1 \; to \; length(SL_{TMR})) \end{aligned} \tag{53}$$

Next, all possible differences between save/restore point indices and store word indices are calculated according to Equation 54. Note that this is different when compared with Equation 36 because this formula must account for the fact that an error cannot be allowed to occur after the TMR to TSR transition or it would be a

TSR error rather than a TMR Type B Error.

$$
\begin{aligned}
&for\ n = 1\ to\ length(SW_{TMR}) \\
&\quad SD_3(n,:) = SW_{TMR}(n) - SI_{TMR}^T \\
&\quad if\ SL_{CTMR}(length(SL_{CTMR})) = P_{loops} \\
&\quad\quad if\ SD_3(n, length(SL_{CTMR})) < 0 \\
&\quad\quad\quad SD_3(n, length(SL_{CTMR})) = 10^6 \\
&\quad\quad end \\
&\quad end \\
&end
\end{aligned}
\tag{54}
$$

Then just as Equation 37 made all values of SD_1 positive, Equation 55 make all values of SD_3 positive as well.

$$SD_4 = SD_3. \cdot (SD_3 > 0) + (SD_3. \cdot (SD_3 < 0) + N_{TMR} \cdot (SD_3 < 0)) \tag{55}$$

The next step is to determine which store word indices minimize the difference between each store word and save index. This is computed in Equation 56. Note that this differs from Equation 38 because there is no second step to determine the absolute minimum distance. This is because it is desirable to determine the early and late scenarios for a Type B-Best error. The absolute minimum of SD_4 might not minimize or maximize the number of instructions computed in TMR mode. Instead, each possible combination of minimum distance from a save index to a store word index is evaluated for total program completion time. The total program completion time for each scenario is then evaluated against the completion times to determine which is slowest (Early) and which is fastest (Late).

$$[a_2, b_2] = min(SD_4) \tag{56}$$

Equations 57, 58, and 59 show how to compute the time to complete each program for each possible combination of minimum distance from a save index to a store word index. Equation 59 is a continuation of Equation 58 because the entire equation could not fit on one page. It also shows that the Type B-Best Early solution is the maximum of these times and the Type B-Best Late solution is the minimum of these times. The $Flag$ variable is used to keep track of whether a particular combination of save index and store word index is allowed. The flag is 1 if the combination is not allowed because the store word following the save index would occur after the TMR to TSR transition. The variable $P_{loops\ TMR\ B\ Best}(n)$ is the new TMR to TSR transition point based on the error location for the n^{th} save index. The variable $n_{CSRP\ B\ Best}(n)$ is the new number of save/restore points to create for the n^{th} save index. The variable $T_{add}(n)$ is the amount of time required to return from the save index to the store word index at which the error occurred for the n^{th} save index. The time to complete the TMR portion of the program for the n^{th} save index is $t_{CTMRBB\ TMR}(n)$. The time to complete the TSR portion of the program for the n^{th} save index is $t_{CTMRBB\ TSR}(n)$. The value NaN is assigned to $t_{CTMRBB\ TMR}(n)$ and $t_{CTMRBB\ TSR}(n)$ when $Flag = 1$ because the max and min functions ignore NaN values and return only numerical values. Finally, $t_{CTMRBBE\ TMR}(n)$, $t_{CTMRBBE\ TSR}(n)$ are the time AHR MIPS spends in TMR and TSR mode when a TMR Type B-Best Early error is encountered. Similarly, $t_{CTMRBBL\ TMR}(n)$, $t_{CTMRBBL\ TSR}(n)$ are the time AHR MIPS spends in TMR and TSR mode when a TMR Type B-Best Late error is encountered.

$for\ n = 1\ to\ length(b_2)$

$Flag = 0$

$if\ SI_{CTMR} = 1$

$P_{loops\ TMR\ B\ Best}(n) = P_{loops}$

$else$

$$P_{loops\ TMR\ B\ Best}(n) = \cdots \left\lceil \frac{SI_{CTMR}(n) + SL_{CTMR} \cdot N_{TMR} + n_{transition} - n_{TMR\_init}}{N_{TMR}} \right\rceil$$

end

$$n_{CSRP\ B\ Best}(n) = \left\lfloor \frac{P_{loops\ TMR\ B\ Best}(n) \cdot N_{TMR} + n_{TMR\_init}}{n_{save}} \right\rfloor \tag{57}$$

$if SI_{CTMR}(n) \leq SW_{CTMR}(b2(n)) - 1$

$T_{add}(n) = \sum_{m=SI_{CTMR}(n)}^{SW_{CTMR}(b2(n))-1} t_{I_{TMR\ m}}$

$elseif\ SL_{CTMR}(n) < P_{loops\ TMR\ B\ Best}(n)$

$T_{add}(n) = \sum_{m=SI_{CTMR}(n)}^{n_{TMR\_init}+N_{TMR}} t_{I_{TMR\ m}} + \sum_{n_{TMR\_init}+1}^{SW_{CTMR}(b2(n))-1} t_{I_{TMR\ m}}$

$else$

$T_{add}(n) = 0$

$Flag = 1$

end

end

$$
\begin{aligned}
&for\ n = 1\ to\ length(b_2)\\
&\quad if\ Flag = 1\\
&\qquad t_{CTMRBB\ TMR}(n) = NaN\\
&\qquad t_{CTMRBB\ TSR}(n) = NaN\\
&\quad else\\
&\qquad t_{nom\ BB} = t_{TMR\ init} + P_{loops\ TMR\ B\ Best}(n) \cdot T_{TMR\ loop} + \cdots\\
&\qquad T_{TMR\ SRP} \cdot n_{CSRP\ B\ Best}(n)\\
&\qquad t_{err\ BB} = T_{TMR\ ttdB} + T_{TMR\ recB} + T_{add}(n)\\
&\qquad if\ P_{loops\ TMR\ B\ Best}(n)\ <\ 250\\
&\qquad\quad t_{CTMRBB\ TMR}(n) = t_{nom\ BB} + t_{err\ BB} + t_{TMR \to TSR}\\
&\qquad\quad t_{CTMRBB\ TSR}(n) = (n_{loops} - P_{loops\ TMR\ B\ Best}(n)) \cdot t_{TSR\ loop} + \cdots\\
&\qquad\quad T_{TSR\ SRP0} + 2 \cdot T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip}\\
&\qquad elseif\ 250\ \leq\ P_{loops\ TMR\ B\ Best}(n)\ <\ 500\\
&\qquad\quad t_{CTMRBB\ TMR}(n) = t_{nom\ BB} + t_{err\ BB} + t_{TMR \to TSR}\\
&\qquad\quad t_{CTMRBB\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Best}(n)) \cdot t_{TSR\ loop} + \cdots\\
&\qquad\quad T_{TSR\ SRP0} + T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip}\\
&\qquad elseif\ 500\ \leq\ P_{loops\ TMR\ B\ Best}(n)\ <\ 750\\
&\qquad\quad t_{CTMRBB\ TMR}(n) = t_{nom\ BB} + t_{err\ BB} + t_{TMR \to TSR}\\
&\qquad\quad t_{CTMRBB\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Best}(n)) \cdot t_{TSR\ loop} + \cdots\\
&\qquad\quad T_{TSR\ SRP1} + T_{TSR\ conc} - \frac{2}{3} T_{TSR\ skip}
\end{aligned}
\tag{58}
$$

$$
\begin{array}{l}
\quad elseif\ 750\ \leq\ P_{loops\ TMR\ B\ Best}(n)\ <\ n_{loops} \\
\quad\quad t_{CTMRBB\ TMR}(n) = t_{nom\ BB} + t_{err\ BB} + t_{TMR \rightarrow TSR} \\
\quad\quad t_{CTMRBB\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Best}(n)) \cdot t_{TSR\ loop} + T_{TSR\ conc} \\
\quad elseif\ P_{loops\ TMR\ B\ Best}(n)\ \geq\ n_{loops} \\
\quad\quad t_{CTMRBB\ TMR}(n) = t_{TMR\ init} + n_{loops} \cdot T_{TMR\ loop} + \cdots \\
\quad\quad T_{TMR\ SRP} \cdot (n_{SRP} - 1) + t_{err\ BB} \\
\quad\quad t_{CTMRBB\ TSR} = 0 \\
\quad end \\
\ end \\
end \\
[T_{CTMR\ B\ Best\ Early}, b_3] = max(t_{CTMRBB\ TMR} + t_{CTMRBB\ TSR}) \\
t_{CTMRBBE\ TMR} = t_{CTMRBB\ TMR}(b_3) \\
t_{CTMRBBE\ TSR} = t_{CTMRBB\ TSR}(b_3) \\
[T_{CTMR\ B\ Best\ Late}, b_4] = min(t_{CTMRBB\ TMR} + t_{CTMRBB\ TSR}) \\
t_{CTMRBBL\ TMR} = t_{CTMRBB\ TMR}(b_4) \\
t_{CTMRBBL\ TSR} = t_{CTMRBB\ TSR}(b_4)
\end{array}
\tag{59}
$$

Remembering that the *min* function used in Equation 56 is defined and used in the same manner as in Equation 38, the same problem with SD_2 possibly being a vector arises for SD_4 as well. This affects the indices used in Equation 57 and Equation 58. If SW_{TMR} is a scalar, Equation 57 is rewritten in Equation 60. If SI_{CTMR} is a scalar, these equations are rewritten in Equations 61, 62, and 63 where Equation 63 is a continuation of Equation 62.

$$
\begin{aligned}
&for\ n = 1\ to\ length(b_2) \\
&\quad Flag = 0 \\
&\quad if\ SI_{CTMR} = 1 \\
&\qquad P_{loops\ TMR\ B\ Best}(n) = P_{loops} \\
&\quad else \\
&\qquad P_{loops\ TMR\ B\ Best}(n) = \cdots \\
&\qquad\quad \left\lceil \frac{SI_{CTMR}(n) + SL_{CTMR} \cdot N_{TMR} + n_{transition} - n_{TMR\_init}}{N_{TMR}} \right\rceil \\
&\quad end \\
&\quad n_{CSRP\ B\ Best}(n) = \left\lfloor \frac{P_{loops\ TMR\ B\ Best}(n) \cdot N_{TMR} + n_{TMR\_init}}{n_{save}} \right\rfloor \qquad (60) \\
&\quad if SI_{CTMR}(n) \leq SW_{CTMR} - 1 \\
&\qquad T_{add}(n) = \textstyle\sum_{m=SI_{CTMR}(n)}^{SW_{CTMR}-1} t_{I_{TMR\ m}} \\
&\quad elseif\ SL_{CTMR}(n) < P_{loops\ TMR\ B\ Best}(n) \\
&\qquad T_{add}(n) = \textstyle\sum_{m=SI_{CTMR}(n)}^{n_{TMR\_init}+N_{TMR}} t_{I_{TMR\ m}} + \sum_{n_{TMR\_init}+1}^{SW_{CTMR}-1} t_{I_{TMR\ m}} \\
&\quad else \\
&\qquad T_{add}(n) = 0 \\
&\qquad Flag = 1 \\
&\quad end \\
&end
\end{aligned}
$$

$$
\begin{aligned}
&Flag = 0 \\
&if\ SI_{CTMR} = 1 \\
&\quad P_{loops\ TMR\ B\ Best} = P_{loops} \\
&else \\
&\quad P_{loops\ TMR\ B\ Best} = \cdots \\
&\qquad \left\lceil \frac{SI_{CTMR} + SL_{CTMR} \cdot N_{TMR} + n_{transition} - n_{TMR\_init}}{N_{TMR}} \right\rceil \\
&end \\
&n_{CSRP\ B\ Best} = \left\lfloor \frac{P_{loops\ TMR\ B\ Best} \cdot N_{TMR} + n_{TMR\_init}}{n_{save}} \right\rfloor \qquad (61) \\
&if SI_{CTMR} \leq SW_{CTMR}(b2) - 1 \\
&\quad T_{add} = \sum_{m=SI_{CTMR}}^{SW_{CTMR}(b2)-1} t_{I_{TMR\ m}} \\
&elseif\ SL_{CTMR}(n) < P_{loops\ TMR\ B\ Best}(n) \\
&\quad T_{add} = \sum_{m=SI_{CTMR}}^{n_{TMR\_init}+N_{TMR}} t_{I_{TMR\ m}} + \sum_{n_{TMR\_init}+1}^{SW_{CTMR}(b2)-1} t_{I_{TMR\ m}} \\
&else \\
&\quad T_{add} = 0 \\
&\quad Flag = 1 \\
&end
\end{aligned}
$$

$$
\begin{aligned}
&if\ Flag = 1 \\
&\quad t_{CTRMBB\ TMR} = NaN \\
&\quad t_{CTRMBB\ TSR} = NaN \\
&else \\
&\quad\quad t_{nom\ BB} = t_{TMR\ init} + P_{loops\ TMR\ B\ Best}(n) \cdot T_{TMR\ loop} + \cdots \\
&\quad\quad T_{TMR\ SRP} \cdot n_{CSRP\ B\ Best}(n) \\
&\quad\quad t_{err\ BB} = T_{TMR\ ttdB} + T_{TMR\ recB} + T_{add}(n) \\
&\quad if\ P_{loops\ TMR\ B\ Best} < 250 \\
&\quad\quad t_{CTRMBB\ TMR} = t_{nom\ BB} + t_{err\ BB} + t_{TMR \rightarrow TSR} \\
&\quad\quad t_{CTRMBB\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Best}) \cdot t_{TSR\ loop} + \cdots \\
&\quad\quad T_{TSR\ SRP0} + 2 \cdot T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip} \\
&\quad elseif\ 250 \leq P_{loops\ TMR\ B\ Best} < 500 \\
&\quad\quad t_{CTRMBB\ TMR} = t_{nom\ BB} + t_{err\ BB} + t_{TMR \rightarrow TSR} \\
&\quad\quad t_{CTRMBB\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Best}) \cdot t_{TSR\ loop} + \cdots \\
&\quad\quad T_{TSR\ SRP0} + T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip} \\
&\quad elseif\ 500 \leq P_{loops\ TMR\ B\ Best} < 750 \\
&\quad\quad t_{CTRMBB\ TMR} = t_{nom\ BB} + t_{err\ BB} + t_{TMR \rightarrow TSR} \\
&\quad\quad t_{CTRMBB\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Best}) \cdot t_{TSR\ loop} + \cdots \\
&\quad\quad T_{TSR\ SRP1} + T_{TSR\ conc} - \frac{2}{3} T_{TSR\ skip} \\
&\quad elseif\ 750 \leq P_{loops\ TMR\ B\ Best} < n_{loops} \\
&\quad\quad t_{CTRMBB\ TMR} = t_{nom\ BB} + t_{err\ BB} + t_{TMR \rightarrow TSR} \\
&\quad\quad t_{CTRMBB\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Best}) \cdot t_{TSR\ loop} + \cdots \\
&\quad\quad T_{TSR\ conc}
\end{aligned}
\tag{62}
$$

$$
\begin{aligned}
&elseif\ P_{loops\ TMR\ B\ Best} \geq n_{loops} \\
&\quad t_{CTRMBB\ TMR} = t_{TMR\ init} + n_{loops} \cdot T_{TMR\ loop} + \cdots \\
&\quad T_{TMR\ SRP} \cdot (n_{SRP} - 1) + t_{err\ BB} \\
&\quad t_{CTRMBB\ TSR} = 0 \\
&end \\
&end \\
&T_{CTMR\ B\ Best\ Early} = t_{CTRMBB\ TMR} + t_{CTRMBB\ TSR} \\
&t_{CTRMBBE\ TMR} = t_{CTRMBB\ TMR} \\
&t_{CTRMBBE\ TSR} = t_{CTRMBB\ TSR} \\
&T_{CTMR\ B\ Best\ Late} = t_{CTRMBB\ TMR} + t_{CTRMBB\ TSR} \\
&t_{CTRMBBL\ TMR} = t_{CTRMBB\ TMR} \\
&t_{CTRMBBL\ TSR} = t_{CTRMBB\ TSR}
\end{aligned}
\tag{63}
$$

AHR MIPS may also encounter TMR MIPS Type B-Worst errors early or late and these are referred to as TMR Type B-Worst Early and TMR Type B-Worst Late errors. As with the TMR Type A Early error, the TMR Type B-Worst Early error has a minimal impact on runtime. As with the TMR Type A Late error, the TMR Type B-Worst Late error is expected to significantly decrease runtime and increase energy usage. The TMR Type B-Worst Early error is shown in Figure 40 while the TMR Type B-Worst Late error is shown in Figure 41.

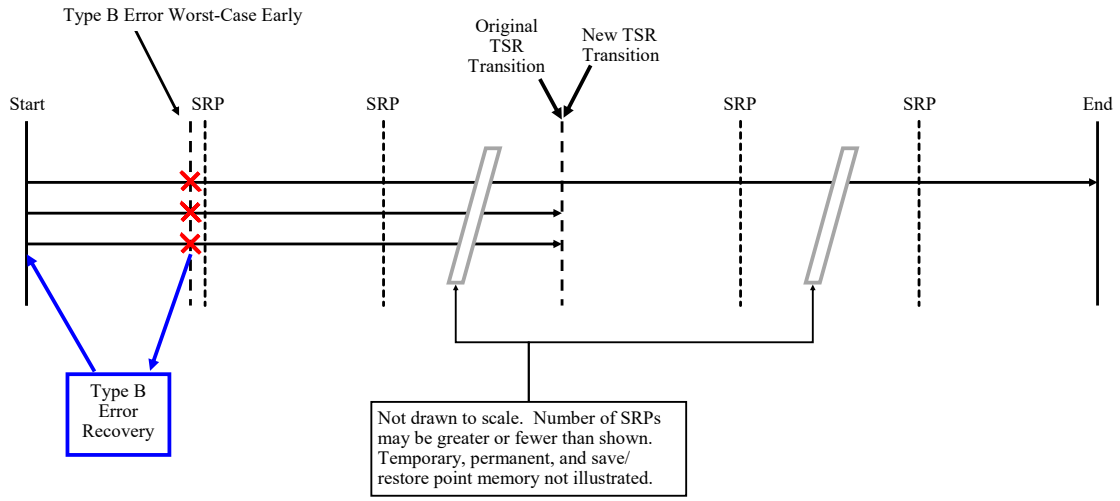

Figure 40. AHR MIPS TMR Type B Worst-Case Early Error Timing Diagram

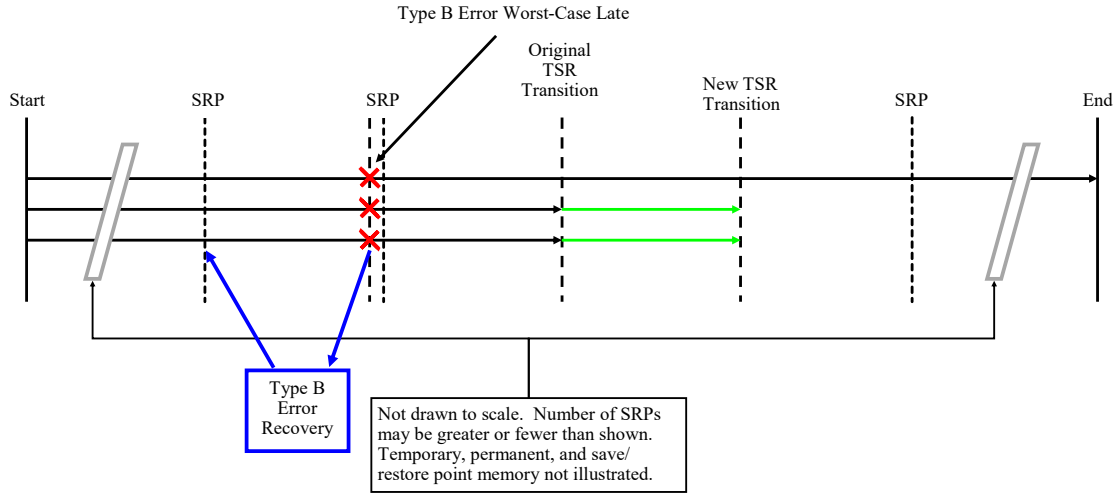

Figure 41. AHR MIPS TMR Type B Worst-Case Late Error Timing Diagram

Equation 64 shows how to compute the time to complete a AHR MIPS program with a TMR Type B-Worst Early error where $P_{loops\ TMR\ B\ Worst\ Early}$ is the number of loops at which the transition point occurs when accounting for the error, $n_{CSRP\ B\ Worst\ Early}$ is the number of save/restore points to create in TMR MIPS when accounting for the error, and $T_{CTMR\ retB\ Worst\ Early}$ is the time needed to return to the point at which the error occurred after recovering from the error. Note that $t_{CTMRBWE\ TMR}$ and $t_{CTMRBWE\ TSR}$ are the time AHR MIPS spends in TMR and

TSR mode respectively when encountering a TMR Type B-Worst Early error.

$$
\begin{aligned}
&t_{nom\ BWE} = t_{TMR\ init} + P_{loops\ TMR\ B\ Worst\ Early} \cdot T_{TMR\ loop} + \cdots \\
&T_{TMR\ SRP} \cdot n_{CSRP\ B\ Worst\ Early} \\
&t_{err\ BWE} = T_{TMR\ SRP\ Err} + T_{TMR\ recB} + T_{CTMR\ retB\ Worst\ Early} \\
&if\ P_{loops\ TMR\ B\ Worst\ Early} < 250 \\
&\quad t_{CTMRBWE\ TMR} = t_{nom\ BWE} + t_{err\ BWE} + t_{TMR \to TSR} \\
&\quad t_{CTMRBWE\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Worst\ Early}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + 2 \cdot T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip} \\
&elseif\ 250 \leq P_{loops\ TMR\ B\ Worst\ Early} < 500 \\
&\quad t_{CTMRBWE\ TMR} = t_{nom\ BWE} + t_{err\ BWE} + t_{TMR \to TSR} \\
&\quad t_{CTMRBWE\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Worst\ Early}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip} \\
&elseif\ 500 \leq P_{loops\ TMR\ B\ Worst\ Early} < 750 \\
&\quad t_{CTMRBWE\ TMR} = t_{nom\ BWE} + t_{err\ BWE} + t_{TMR \to TSR} \\
&\quad t_{CTMRBWE\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Worst\ Early}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP1} + T_{TSR\ conc} - \frac{2}{3} T_{TSR\ skip} \\
&elseif\ 750 \leq P_{loops\ TMR\ B\ Worst\ Early} < n_{loops} \\
&\quad t_{CTMRBWE\ TMR} = t_{nom\ BWE} + t_{err\ BWE} + t_{TMR \to TSR} \\
&\quad t_{CTMRBWE\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Worst\ Early}) \cdot t_{TSR\ loop} + T_{TSR\ conc} \\
&elseif\ P_{loops\ TMR\ B\ Worst\ Early} \geq n_{loops} \\
&\quad t_{CTMRBWE\ TMR} = t_{TMR\ init} + n_{loops} \cdot T_{TMR\ loop} + \cdots \\
&\quad T_{TMR\ SRP} \cdot (n_{SRP} - 1) + t_{err\ BWE} \quad t_{CTMRBWE\ TSR} = 0 \\
&end \\
&T_{CTMR\ B\ Worst\ Early} = t_{CTMRBWE\ TMR} + t_{CTMRBWE\ TSR}
\end{aligned}
\tag{64}
$$

The time $T_{CTMR\ retB\ Worst\ Early}$ is computed according to Equation 65 where SDT_{CTMR} is the save time difference between consecutive save points and $W_{SI_{CTMR}}$ is the index of the worst-case SDT_{CTMR}. This is nearly identical to Equation 43.

$$\begin{aligned} SDT_{CTMR} &= ST_{CTMR} - [0, ST_{CTMR}(1\ to\ length(ST_{CTMR}) - 1)] \\ T_{CTMR\ retB\ Worst\ Early} &= SDT_{CTMR}(2) \end{aligned} \tag{65}$$

Next, the loop count at which the TMR to TSR transition will occur after encountering an error is determined using Equation 66.

$$\begin{aligned} &if\ SL_{CTMR}(1) = 0 \\ &\quad P_{loops\ TMR\ B\ Worst\ Early} = P_{loops} \\ &else \\ &\quad P_{loops\ TMR\ B\ Worst\ Early} = \cdots \\ &\left\lceil \frac{SI_{CTMR}(1) + SL_{CTMR}(1) \cdot N_{TMR} + n_{transition} - n_{TMR\_init}}{N_{TMR}} \right\rceil \\ &end \end{aligned} \tag{66}$$

And finally, $n_{CSRP\ B\ Worst\ Early}$ is determined according to Equation 67.

$$n_{CSRP\ B\ Worst\ Early} = \left\lfloor \frac{P_{loops\ TMR\ B\ Worst\ Early} \cdot N_{TMR} + n_{TMR\_init}}{n_{save}} \right\rfloor \tag{67}$$

Equation 68 shows how to compute the time to complete a AHR MIPS program with a TMR Type B-Worst Late error where $P_{loops\ TMR\ B\ Worst\ Late}$ is the number of loops at which the transition point occurs when accounting for the error, $n_{CSRP\ B\ Worst\ Late}$ is the number of save/restore points to create in TMR MIPS when accounting for the error, and $T_{CTMR\ retB\ Worst\ Late}$ is the time needed to return to the point at which the error occurred after recovering from the error. Note that $t_{CTMRBWL\ TMR}$ and $t_{CTMRBWL\ TSR}$ are the time AHR MIPS spends in TMR and TSR mode respectively when encountering a TMR Type B-Worst Late error.

$$
\begin{aligned}
&t_{nom\ BWL} = t_{TMR\ init} + P_{loops\ TMR\ B\ Worst\ Late} \cdot T_{TMR\ loop} + \cdots \\
&T_{TMR\ SRP} \cdot n_{CSRP\ B\ Worst\ Late} \\
&t_{err\ BWL} = T_{TMR\ SRP\ Err} + T_{TMR\ recB} + T_{CTMR\ retB\ Worst\ Early} \\
&if\ P_{loops\ TMR\ B\ Worst\ Late} < 250 \\
&\quad t_{CTMRBWL\ TMR} = t_{nom\ BWE} + t_{err\ BWE} + t_{TMR \rightarrow TSR} \\
&\quad t_{CTMRBWL\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Worst\ Late}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + 2 \cdot T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip} \\
&elseif\ 250 \leq P_{loops\ TMR\ B\ Worst\ Late} < 500 \\
&\quad t_{CTMRBWL\ TMR} = t_{nom\ BWE} + t_{err\ BWE} + t_{TMR \rightarrow TSR} \\
&\quad t_{CTMRBWL\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Worst\ Late}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP0} + T_{TSR\ SRP1} + T_{TSR\ conc} - T_{TSR\ skip} \\
&elseif\ 500 \leq P_{loops\ TMR\ B\ Worst\ Late} < 750 \\
&\quad t_{CTMRBWL\ TMR} = t_{nom\ BWE} + t_{err\ BWE} + t_{TMR \rightarrow TSR} \\
&\quad t_{CTMRBWL\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Worst\ Late}) \cdot t_{TSR\ loop} + \cdots \\
&\quad T_{TSR\ SRP1} + T_{TSR\ conc} - \frac{2}{3} T_{TSR\ skip} \\
&elseif\ 750 \leq P_{loops\ TMR\ B\ Worst\ Late} < n_{loops} \\
&\quad t_{CTMRBWL\ TMR} = t_{nom\ BWE} + t_{err\ BWE} + t_{TMR \rightarrow TSR} \\
&\quad t_{CTMRBWL\ TSR} = (n_{loops} - P_{loops\ TMR\ B\ Worst\ Late}) \cdot t_{TSR\ loop} + T_{TSR\ conc} \\
&elseif\ P_{loops\ TMR\ B\ Worst\ Late} \geq n_{loops} \\
&\quad t_{CTMRBWL\ TMR} = t_{TMR\ init} + n_{loops} \cdot T_{TMR\ loop} + \cdots \\
&\quad T_{TMR\ SRP} \cdot (n_{SRP} - 1) + t_{err\ BWE} \\
&\quad t_{CTMRBWL\ TSR} = 0 \\
&end \\
&T_{CTMR\ B\ Worst\ Late} = t_{CTMRBWL\ TMR} + t_{CTMRBWL\ TSR}
\end{aligned}
\tag{68}
$$

The time $T_{CTMR\ retB\ Worst\ Late}$ is computed according to Equation 69. This is nearly identical to Equation 43.

$$T_{CTMR\ retB\ Worst\ Late} = SDT_{CTMR}(length(SDT_{CTMR})) \tag{69}$$

Next, the loop count at which the TMR to TSR transition will occur after encountering an error is determined using Equation 70.

$$\begin{aligned}
&if\ SL_{CTMR}(length(SDT_{CTMR}) - 1) = 0 \\
&\quad P_{loops\ TMR\ B\ Worst\ Late} = P_{loops} \\
&else \\
&\quad P_{loops\ TMR\ B\ Worst\ Late} = \cdots \\
&\left\lceil \left(SI_{CTMR}(length(SDT_{CTMR}) - 1) + \cdots \right.\right. \\
&SL_{CTMR}(length(SDT_{CTMR}) - 1) \cdot N_{TMR} + \cdots \\
&\left.\left. n_{transition} - n_{TMR_init} \right) \Big/ N_{TMR} \right\rceil \\
&end
\end{aligned} \tag{70}$$

And finally, $n_{CSRP\ B\ Worst\ Late}$ is determined according to Equation 71.

$$n_{CSRP\ B\ Worst\ Late} = \left\lfloor \frac{P_{loops\ TMR\ B\ Worst\ Late} \cdot N_{TMR} + n_{TMR\_init}}{n_{save}} \right\rfloor \tag{71}$$

In contrast to the TMR errors which can affect the TMR to TSR transition point, TSR errors do not affect the transition point; however, TSR worst-case errors may be affected by the transition point. The best-case errors are unaffected by the transition point.

When AHR MIPS encounters a TSR Best-case error, it encounters it immediately after the creation of a save/restore point. This could be the save/restore point created by the transition from TMR to TSR, or any of the save/restore points created by TSR

MIPS after AHR MIPS enters TSR mode. Regardless of where the which save/restore point the TSR Best-case error occurs after, the recovery time is always the same. This is because of the way the TSR MIPS Best-case error was defined to be injected immediately prior to the branch comparison instruction before the first store word instruction after creating a save/restore point. A few examples of AHR MIPS TSR Best-case errors are shown in Figures 42, 43, 44, and 45 where the transition occurs before the first, second, or third TSR save/restore creation point or after the third TSR save/restore creation point respectively.

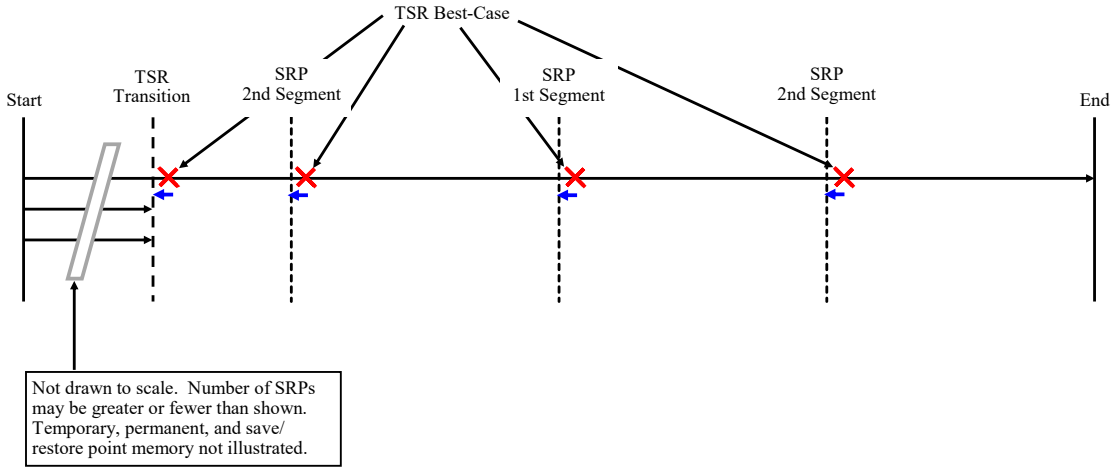

Figure 42. AHR MIPS TSR Best-Case Early Error Timing Diagram 1

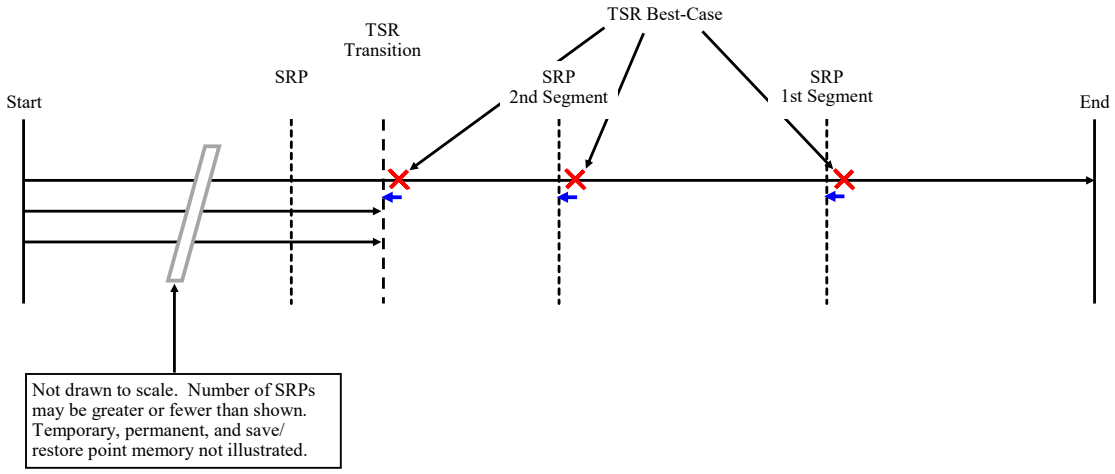

Figure 43. AHR MIPS TSR Best-Case Early Error Timing Diagram 2

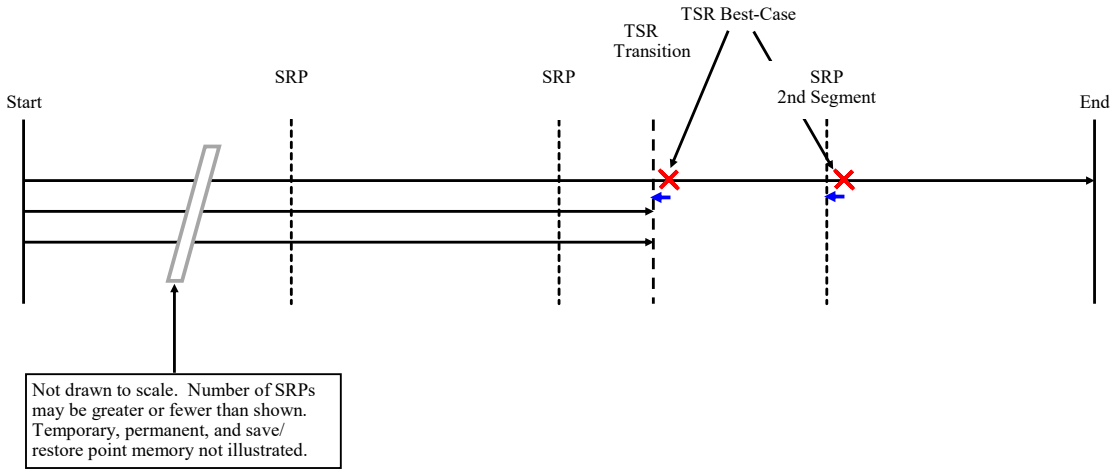

Figure 44. AHR MIPS TSR Best-Case Early Error Timing Diagram 3

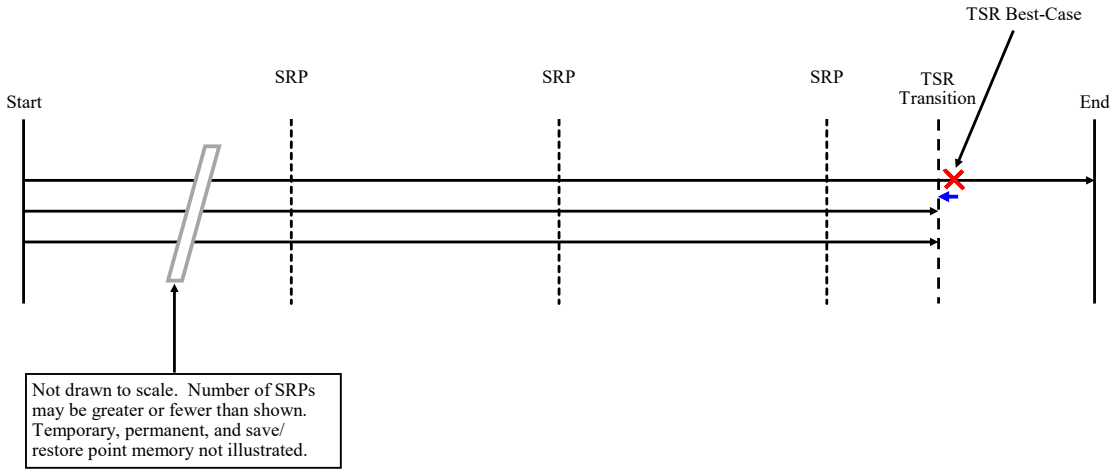

Figure 45. AHR MIPS TSR Best-Case Early Error Timing Diagram 4

The time needed to complete a AHR MIPS program experiencing a TSR Best-case error is given in Equation 72 where $T_{TSR\ Rec}$ and $T_{TSR\ ret}$ were previously defined in Equation 44.

$$\begin{aligned} T_{CTSR\ Best} &= T_{AHR\ MIPS} + T_{TSR\ Rec} + T_{TSR\ ret} \\ T_{CTSR\ Best} &= t_{AHR\ TMR} + t_{AHR\ TSR} + T_{TSR\ Rec} + T_{TSR\ ret} \\ t_{CTSRB\ TMR} &= t_{AHR\ TMR} \\ t_{CTSRB\ TSR} &= t_{AHR\ TSR} + T_{TSR\ Rec} + T_{TSR\ ret} \\ T_{CTSR\ Best} &= t_{CTSRB\ TMR} + t_{CTSRB\ TSR} \end{aligned} \tag{72}$$

TSR Worst-case errors in AHR MIPS require special attention. While TSR Worst-case errors in TSR MIPS take place at the end of creating a save/restore point in the second save/restore point memory segment, that may not be possible in AHR MIPS depending on when the TMR to TSR transition takes place. If that transition occurs before the first TSR MIPS save/restore point is created, then the TSR worst-case error is still encountered at the end of creating a save/restore point in the second segment; in this case this would be the save/restore point created when the loop counter is at 250. This scenario is shown in Figure 46.

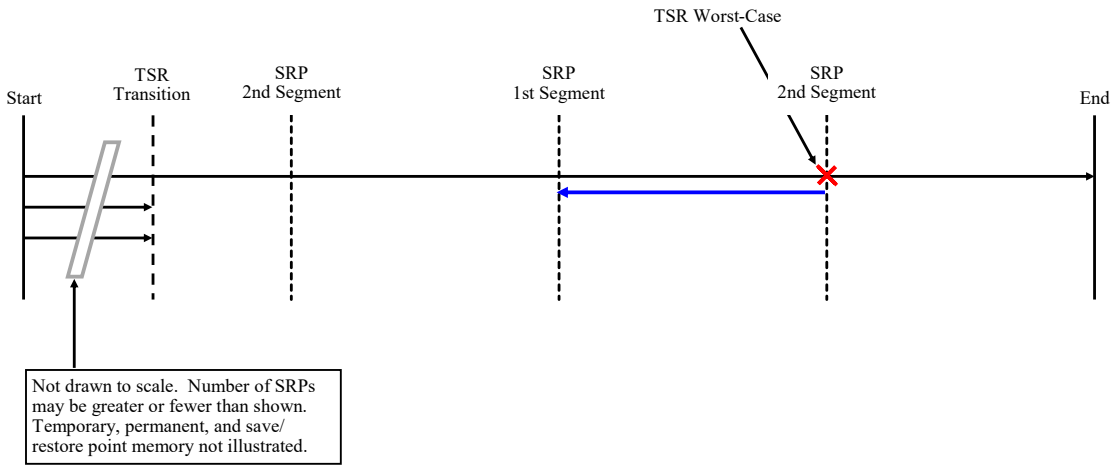

Figure 46. AHR MIPS TSR Worst-Case Early Error Timing Diagram 1

When the TMR to TSR transition occurs after what would have been the first TSR MIPS save/restore point creation and before the second TSR MIPS save/restore point creation, there are two possibilities for a worst-case error. These possibilities are shown in Figure 47. Note that the first save/restore point created after the transition is always to the second save/restore point memory segment. This means that an error at the end of this save/restore point creation may not be the worst-case error. The worst-case error may be the one that occurs at the end of the next save/restore point creation which saves to the first save/restore point memory segment. The time to recover from the error and return to the point at which the error was encountered is calculated for both of these scenarios and the one that takes longer is the worst-case error.

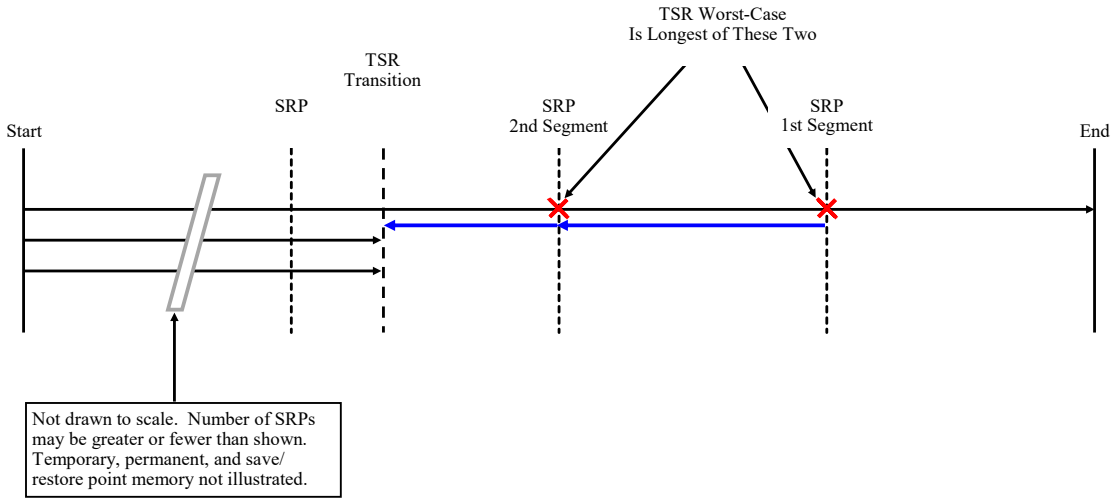

Figure 47. AHR MIPS TSR Worst-Case Early Error Timing Diagram 2

If the TSR Worst-case error occurs after the second TSR MIPS save/restore point creation and before the third, then it is unclear what the worst-case error might be. According to the original definition of a TSR MIPS Worst-case error, it is an error that maximizes the number of instructions that TSR MIPS must re-execute. Therefore, the error may occur at the end of creating the third TSR MIPS save/restore point or at the last branch comparison at the end of the program. The amount of time to return to the point at which the error occurred is calculated for both scenarios, and the one that takes longer is the worst-case scenario. This is illustrated graphically in Figure 48.

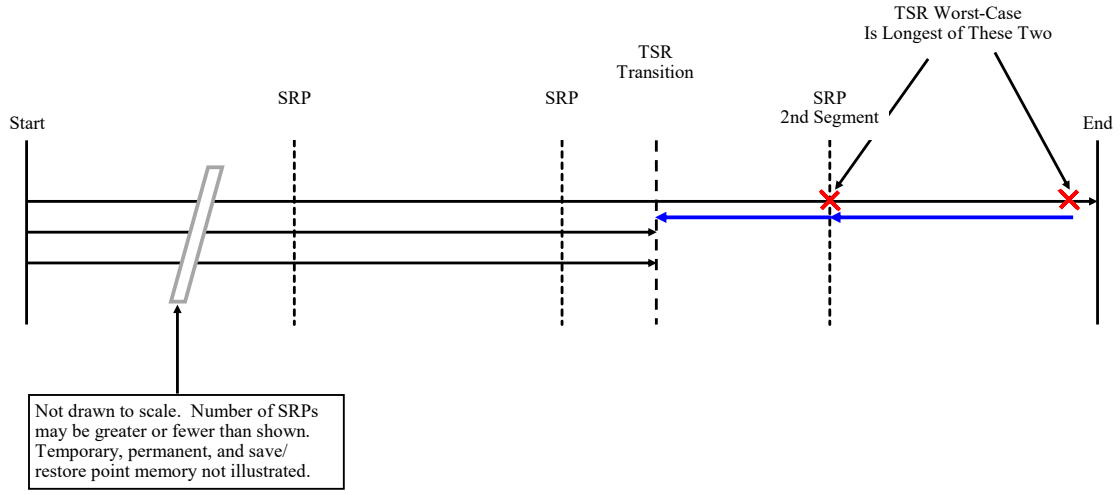

Figure 48. AHR MIPS TSR Worst-Case Early Error Timing Diagram 3

Finally, if the TSR Worst-case error occurs after the last TSR MIPS save/restore point creation, the worst-case error occurs at the last branch comparison at the end of the program as shown in Figure 49.

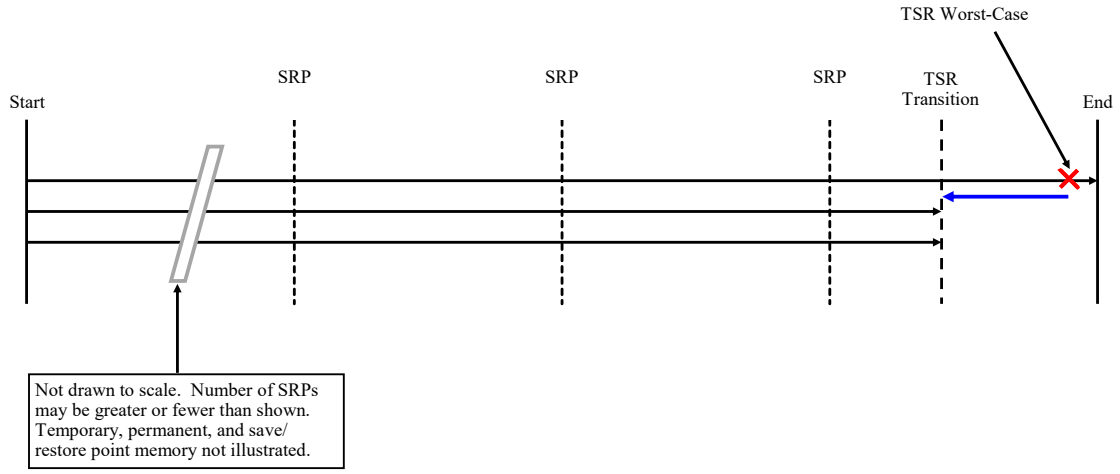

Figure 49. AHR MIPS TSR Worst-Case Early Error Timing Diagram 4

No errors are injected to Basic MIPS because it has no way of detecting or correcting the errors. Any errors injected into a register to be stored to memory would not impact the runtime or energy usage of Basic MIPS.

Equation 73 and Equation 74 show how to compute the time to complete a AHR

MIPS program experiencing a TSR Worst-case error where Equation 74 is a continuation of Equation 73. If the transition point occurs before the completion of the first 250 loops, the AHR MIPS TSR worst-case error is identical to the TSR MIPS worst-case error in that the added time to complete the program is the same as in Equation 46.

If the transition point occurs between the completion of 250 loops and 500 loops, there are two possibilities for the worst-case error. The first is that the error occurs at the end of creating the save/restore point upon completion of 500 loops, in which case all loops after the TMR to TSR transition must be re-completed and the save/restore point must be completed without error as well ($ctsrw_1$). The second is that the error occurs at the end of creating the save/restore point upon completion of 750 loops, in which case all loops after previous save/restore point creation must be re-completed and the save/restore point at loop number 750 must be completed without error as well ($ctsrw_2$).

If the transition point occurs between the completion of 500 loops and 750 loops, there are two possibilities for the worst-case error. The first is that the error occurs at the end of creating the save/restore point upon completion of 750 loops, in which case all loops after the TMR to TSR transition must be re-completed and the save/restore point must be completed without error as well ($ctsrw_3$). The second is that the error occurs at the last store word instruction in the program and the nearly 250 complete loops since the creation of the save/restore point at loop 750 must be re-completed ($ctsrw_4$). The only way to know which takes longer to complete is to calculate the values for both, compare the results, and select the larger of the two. If the transition point occurs after the completion of 750 loops, the worst-case error occurs at the last store word at the end of the program and all loops from the TMR to TSR transition to the end of the program must be re-completed.

Note that $t_{CTSRW\ TMR}$ and $t_{CTSRW\ TSR}$ are the time AHR MIPS spends in TMR and TSR mode respectively when encountering a TSR Worst-case error.

$$
\begin{aligned}
&if\ P_{loops}\ <\ 250\\
&\quad t_{CTSRW\ TMR} = t_{AHR\ TMR}\\
&\quad t_{CTSRW\ TSR} = t_{AHR\ TSR} + T_{TSR\ Rec} + \textstyle\sum_{n=N_{TSR}-3}^{N_{TSR}} t_{I_{TSR\ n}} + \cdots\\
&\quad 250 \cdot T_{TSR\ loop} + T_{TSR\ SRP1\ Err}\\
&elseif\ 250\ \leq\ P_{loops}\ <\ 500\\
&\quad ctsrw_1 = T_{TSR\ Rec} + \textstyle\sum_{n=N_{TSR}-3}^{N_{TSR}} t_{I_{TSR\ n}} + (500 - P_{loops}) \cdot T_{TSR\ loop} + \cdots\\
&\quad T_{TSR\ SRP1\ Err}\\
&\quad ctsrw_2 = T_{TSR\ Rec} + \textstyle\sum_{n=N_{TSR}-3}^{N_{TSR}} t_{I_{TSR\ n}} + 250 \cdot T_{TSR\ loop} + T_{TSR\ SRP0\ Err}\\
&\quad t_{CTSRW\ TMR} = t_{AHR\ TMR}\\
&\quad if\ ctsrw_1 > ctsrw_2\\
&\qquad t_{CTSRW\ TSR} = t_{AHR\ TSR} + ctsrw_1\\
&\quad else\\
&\qquad t_{CTSRW\ TSR} = t_{AHR\ TSR} + ctsrw_2\\
&\quad end
\end{aligned}
\tag{73}
$$

$$
\begin{aligned}
&elseif\ 500\ \leq\ P_{loops}\ <\ 750\\
&\quad ctsrw_3 = T_{TSR\ Rec} + \textstyle\sum_{n=N_{TSR}-3}^{N_{TSR}} t_{I_{TSR\ n}} + \cdots\\
&\quad (750 - P_{loops}) \cdot T_{TSR\ loop} + T_{TSR\ SRP1\ Err}\\
&\quad ctsrw_4 = T_{TSR\ Rec} + \textstyle\sum_{n=N_{TSR}-3}^{N_{TSR}} t_{I_{TSR\ n}} + 249 \cdot T_{TSR\ loop} + \cdots\\
&\quad \textstyle\sum_{n_{TSR\ init}+1}^{SW_{TSR}(length(SW_{TSR})-1)} t_{I_{TSR\ n}}\\
&\qquad t_{CTSRW\ TMR} = t_{AHR\ TMR}\\
&\quad if\ ctsrw_3 > ctsrw_4\\
&\qquad t_{CTSRW\ TSR} = t_{AHR\ TSR} + ctsrw_3\\
&\quad else\\
&\qquad t_{CTSRW\ TSR} = t_{AHR\ TSR} + ctsrw_4\\
&\quad end\\
&elseif\ P_{loops}\ \geq\ 750\\
&\quad t_{CTSRW\ TMR} = t_{AHR\ TMR}\\
&\quad t_{CTSRW\ TSR} = t_{AHR\ TSR} + T_{TSR\ Rec} + \textstyle\sum_{n=N_{TSR}-3}^{N_{TSR}} t_{I_{TSR\ n}} + \cdots\\
&\quad (n_{loops} - P_{loops} - 1) \cdot T_{TSR\ loop} + \textstyle\sum_{n_{TSR\ init}+1}^{SW_{TSR}(length(SW_{TSR})-1)} t_{I_{TSR\ n}}\\
&end\\
&T_{CTSR\ Worst} = t_{CTSRW\ TMR} + t_{CTSRW\ TSR}
\end{aligned}
\tag{74}
$$

5.4.2 Energy Calculations

Now that the timing computations are complete, the energy computations must be performed. These computations are straightforward for TMR MIPS and TSR MIPS programs even when errors are injected. The time to complete these programs is multiplied by the dynamic power used by the appropriate architecture. The TMR Type A, Type B-Best, and Type B-Worst error energy calculations are shown in Equations 75, 76, and 77 respectively. The TSR MIPS Best-case and Worst-case error energy calculations are shown in Equations 78, and 79 respectively.

$$E_{TMR\ ErrA} = P_{TMR\ MIPS} \cdot T_{TMR\ ErrA} \tag{75}$$

$$E_{TMR\ ErrB\ Best} = P_{TMR\ MIPS} \cdot T_{TMR\ ErrB\ Best} \tag{76}$$

$$E_{TMR\ ErrB\ Worst} = P_{TMR\ MIPS} \cdot T_{TMR\ ErrB\ Worst} \tag{77}$$

$$E_{TSR\ Best} = P_{TSR\ MIPS} \cdot T_{TSR\ Best} \tag{78}$$

$$E_{TSR\ Worst} = P_{TSR\ MIPS} \cdot T_{TSR\ Worst} \tag{79}$$

While it was trivial to calculate the energy used by TMR MIPS and TSR MIPS programs experiencing errors, it is more complicated to calculate the energy used by programs running in AHR MIPS. It is more difficult because of the time divided between TMR and TSR modes of operation. Fortunately, the times to complete the TMR and TSR portions were recorded separately to make these calculations simpler.

Equations 80 and 81 show how to calculate the energy used by AHR MIPS when encountering TMR Type A Early and Late errors respectively.

$$E_{CTMR\ A\ Early} = P_{CTMR\ MIPS} \cdot t_{CTMRAE\ TMR} + P_{CTSR\_MIPS} \cdot t_{CTMRAE\ TSR} \tag{80}$$

$$E_{CTMR\ A\ Late} = P_{CTMR\ MIPS} \cdot t_{CTMRAL\ TMR} + P_{CTSR\_MIPS} \cdot t_{CTMRAL\ TSR} \tag{81}$$

Equations 82 and 83 show how to calculate the energy used by AHR MIPS when encountering a TMR Type B-Best Early and Late error respectively.

$$E_{CTMR\ B\ Best\ Early} = P_{CTMR\ MIPS} \cdot t_{CTMRBBE\ TMR} + \cdots \\ P_{CTSR\_MIPS} \cdot t_{CTMRBBE\ TSR} \tag{82}$$

$$E_{CTMR\ B\ Best\ Late} = P_{CTMR\ MIPS} \cdot t_{CTMRBBL\ TMR} + \cdots \\ P_{CTSR\_MIPS} \cdot t_{CTMRBBL\ TSR} \tag{83}$$

Equations 84 and 85 show how to calculate the energy used by AHR MIPS when encountering a TMR Type B-Worst Early and Late error respectively.

$$E_{CTMR\ B\ Worst\ Early} = P_{CTMR\ MIPS} \cdot t_{CTMRBWE\ TMR} + \cdots \\ P_{CTSR\_MIPS} \cdot t_{CTMRBWE\ TSR} \tag{84}$$

$$E_{CTMR\ B\ Worst\ Late} = P_{CTMR\ MIPS} \cdot t_{CTMRBWL\ TMR} + \cdots \\ P_{CTSR\_MIPS} \cdot t_{CTMRBWL\ TSR} \tag{85}$$

Equations 86 and 87 show how to calculate the energy used by AHR MIPS when encountering a TSR Type Best-Case and Worst-Case error respectively.

$$E_{CTSR\ Best} = P_{CTMR\ MIPS} \cdot t_{CTSRB\ TMR} + P_{CTSR\_MIPS} \cdot t_{CTSRB\ TSR} \tag{86}$$

$$E_{CTSR\ Worst} = P_{CTMR\ MIPS} \cdot t_{CTSRW\ TMR} + P_{CTSR\_MIPS} \cdot t_{CTSRW\ TSR} \tag{87}$$

5.5 HITL Simulation with Error Injection

During HITL simulation with error injection, all of the aforementioned best-case and worst-case scenarios would be implemented in random selection of the 1,000 programs to determine how much energy was consumed by the various architectures when handling the errors. These experiments would also allow for current, voltage, and duration measurements that would enable energy consumption calculations as previously performed for the error free HITL simulations in Section 4.5. As before, a DE10-Standard energy usage baseline will be established immediately before each individual experiment so that the baseline energy usage may be subtracted from the energy used while conducting an individual HITL simulation with error injection experiment.

The error injection in HITL is performed by using the same VHDL code used to inject errors in the software simulations. This VHDL code, which was simulated in Mentor Graphics QuestaSim, is also easily implemented in hardware by the Quartus software used to program the Cyclone V FPGA. However, because it will be implemented in hardware, no modifications need to be made to accommodate the computer resource limitations encountered with software simulations. This means that each Basic MIPS, TMR MIPS, TSR MIPS, and AHR MIPS program will run in its entirety with errors present.

Due to the aforementioned hardware problems which prevented HITL simulation measurements of processor energy usage, no HITL energy results are available for HITL simulation with error injection. However, some HITL timing results are available for error injection into TMR and TSR MIPS. These HITL simulations with error injection to measure timing were accomplished using the same methods presented in Section 4.5.1, but make use of the Terasic DE-10 Standard rather than the Terasic SoCKit.

5.6 Summary

This chapter discussed the rate at which errors should be injected as well as the time, location, and method to inject errors. This chapter also presented the analytical framework necessary to compute the time and energy required to complete programs in TMR, TSR, and AHR MIPS in the presence of errors. Finally, this chapter touched on the method to perform HITL simulations with error injection. The next chapter will discuss the results of the error injection simulations and analyses.

VI. Error Injection Analysis and Results

6.1 Introduction

This chapter discusses the results of the error injection software simulations, analyses, and hardware-in-the-loop simulations discussed in Chapter V, Sections 5.4 and 5.5. This chapter provides analyses which demonstrate Adaptive-Hybrid Redundancy's (AHR's) ability to provide space vehicle designers, mission planners, and operators the flexibility to optimize processing speed and energy usage according to mission needs and error conditions.

Section 6.2 discusses the results of simulations and analyses that were performed with error injection; these simulations and analyses were first discussed in Section 5.4. This section also evaluates AHR's performance against Triple Modular Redundancy (TMR) and Temporal Software Redundancy (TSR) in the presence of errors; using the error free unmitigated approach as a baseline. Section 6.3 discusses the work done on HITL simulations with error injection first discussed in Section 5.5.

6.2 Software Simulation with Error Injection

This section examines the results of software simulations and computational analysis when errors are injected as described in Section 5.4. Only one error is injected into each program. The types of errors injected into TMR MIPS programs are TMR Type A and B errors. TMR Type A errors occur when a single error is injected into a register that will be stored to memory. A TMR Type B error occurs when different errors are injected into a register that will be stored to memory of two different processors. TMR Type B errors are further divided into best-case (TMR Type B-Best) and worst-case (TMR Type B-Worst) errors. TMR Type B-Best errors minimize the amount of rework needed to recover to the point at which the error initially occurred

while a TMR Type B-Worst error maximizes the same. A TSR error is an error injected into one of a pair of duplicate registers that will be stored to memory. TSR errors are also divided into best-case and worst-case errors in the same way as TMR Type B errors. AHR MIPS can experience either TMR or TSR errors. For AHR MIPS, TMR Type A, TMR Type B-Best, and TMR Type B-Worst errors are further subdivided into early and late errors. An early error occurs early during a program's execution and does not significantly impact the TMR to TSR transition point or the program's overall runtime when compared to no error occurring. A late error occurs close to the error free TMR to TSR transition point and causes the transition point to move 15,000 instructions away from the point at which error recovery resumes, assuming the TMR to TSR transition point occurs after 15,000 error free instructions. This causes the program to run in TMR MIPS for significantly more instructions than in the error free case. The result is that the program runs faster and uses much more energy when a late error is experienced than when an early error of the same type (A, B-Best, B-Worst) or no error is encountered. Review Section 5.4.1 and Figures 31 to 49 for more details.

This section first looks at TMR MIPS errors, then TSR MIPS errors, and finally AHR MIPS errors.

6.2.1 TMR MIPS Error Injection Results

As discussed in Section 5.4, TMR MIPS errors are divided into Type A and Type B errors. The results of Type A, Type B-Best, and Type B-Worst error injection for the 1,000 programs discussed in Section 4.4 are shown in Figure 50. This figure shows the time and energy needed to complete each program for each TMR error type as well as the average time to complete a program for each error type (including no error). The differences between Type A errors, Type B-Best errors, and no errors

results are indiscernible in this figure. This result meets expectations because Type A errors and Type B-Best errors were chosen to minimize the impact on TMR MIPS performance. Type B-Worst errors significantly increase the time and energy required to complete the program. This result also meets expectations because Type B-Worst errors were chosen to maximize the impact on TMR MIPS performance.

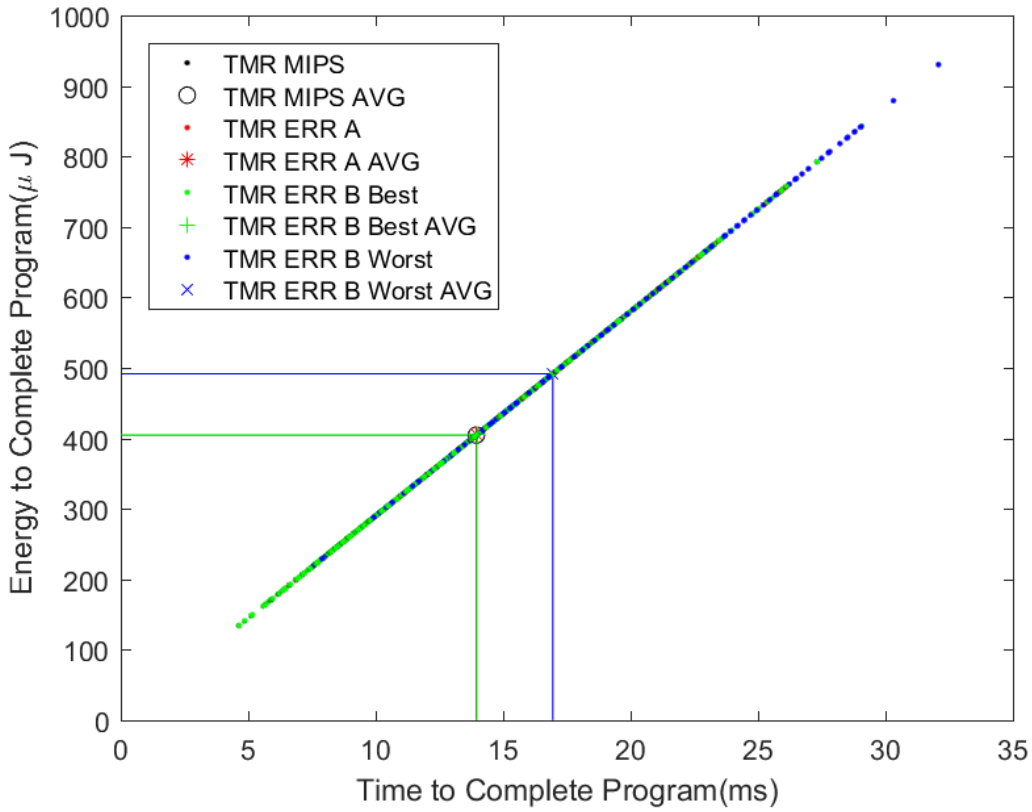

Figure 50. Software Simulation of TMR MIPS Errors - Energy vs. Time to Complete

6.2.2 TSR MIPS Error Injection Results

TSR MIPS errors are divided into best-case and worst-case errors. The results of TSR MIPS error injection for the 1,000 programs discussed in Section 4.4 are shown in Figure 51. This figure shows the time and energy needed to complete each program for each TSR error type as well as the average time to complete a program for each

error type (including no error). The differences between best-case errors and no errors results are indiscernible in this figure. This result meets expectations because best-case errors were chosen to minimize the impact on TSR MIPS performance. Worst-case errors significantly increase the time and energy required to complete the program. This result also meets expectations because worst-case errors were chosen to maximize the impact on TSR MIPS performance.

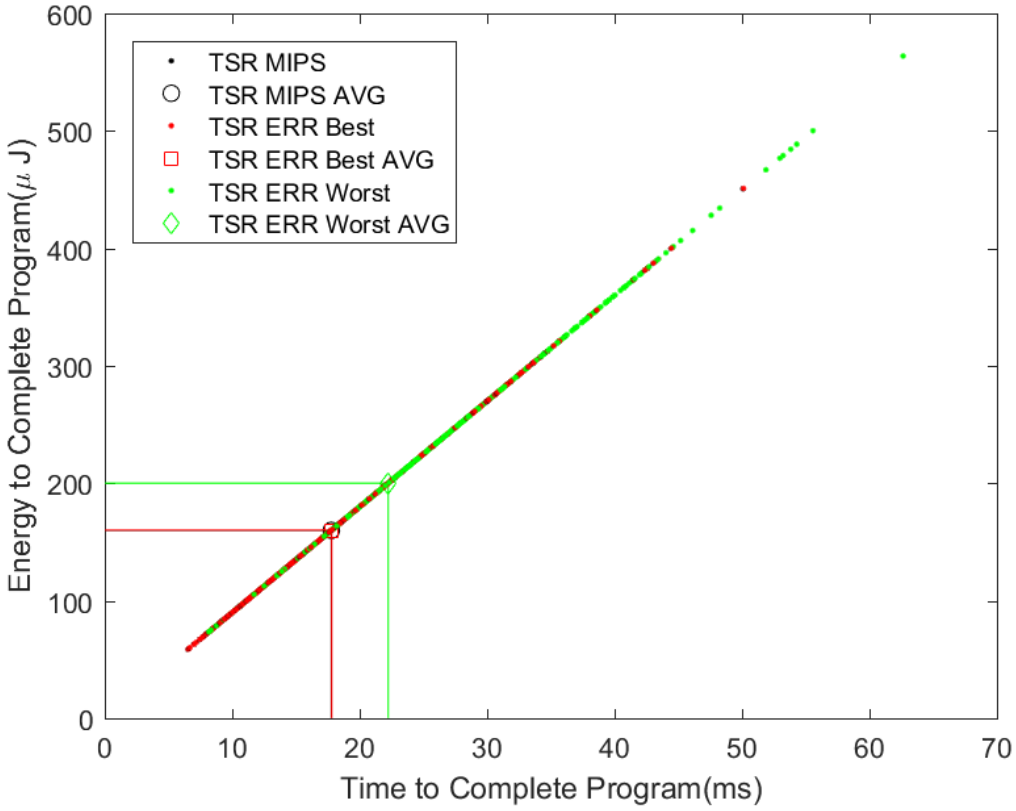

Figure 51. Software Simulation of TSR MIPS Errors - Energy vs. Time to Complete

6.2.3 AHR MIPS Error Injection Results

AHR MIPS errors are divided into 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 Best, and TSR Worst errors. The results of AHR MIPS error

injection for the 1,000 programs using a TMR to TSR transition point of 15,000 instructions discussed in Section 4.4 are shown in Figure 52. The differences between TMR Type A Early, TMR Type B-Best Early, TSR Best, and no errors results are indiscernible in this figure. This result meets expectations because these errors were chosen to minimize the impact on AHR MIPS performance. The TMR Type A Late and TMR Type B-Best Late errors use more energy than AHR MIPS with no errors, but complete in less time. This is because the late errors cause AHR MIPS to remain in TMR mode longer than if no error occurred. This result meets expectations because TMR Type A Late errors and TMR Type B-Best Late errors were chosen to maximize time spent in TMR and minimize total program run time. The TMR Type B-Worst Early, TMR Type B-Worst Late, and TSR Worst errors use more energy and take more time to complete than AHR MIPS with no errors. These results meet expectations because TMR Type B-Worst and TSR Worst errors cause the maximum amount of program re-computation (must re-compute more instructions than any other error type). It was also expected that AHR MIPS would recover from TMR Type B-Worst errors more quickly than TSR Worst errors, but require more energy to do so. This becomes even more evident when viewing only the averages of all 1,000 programs in Figure 53.

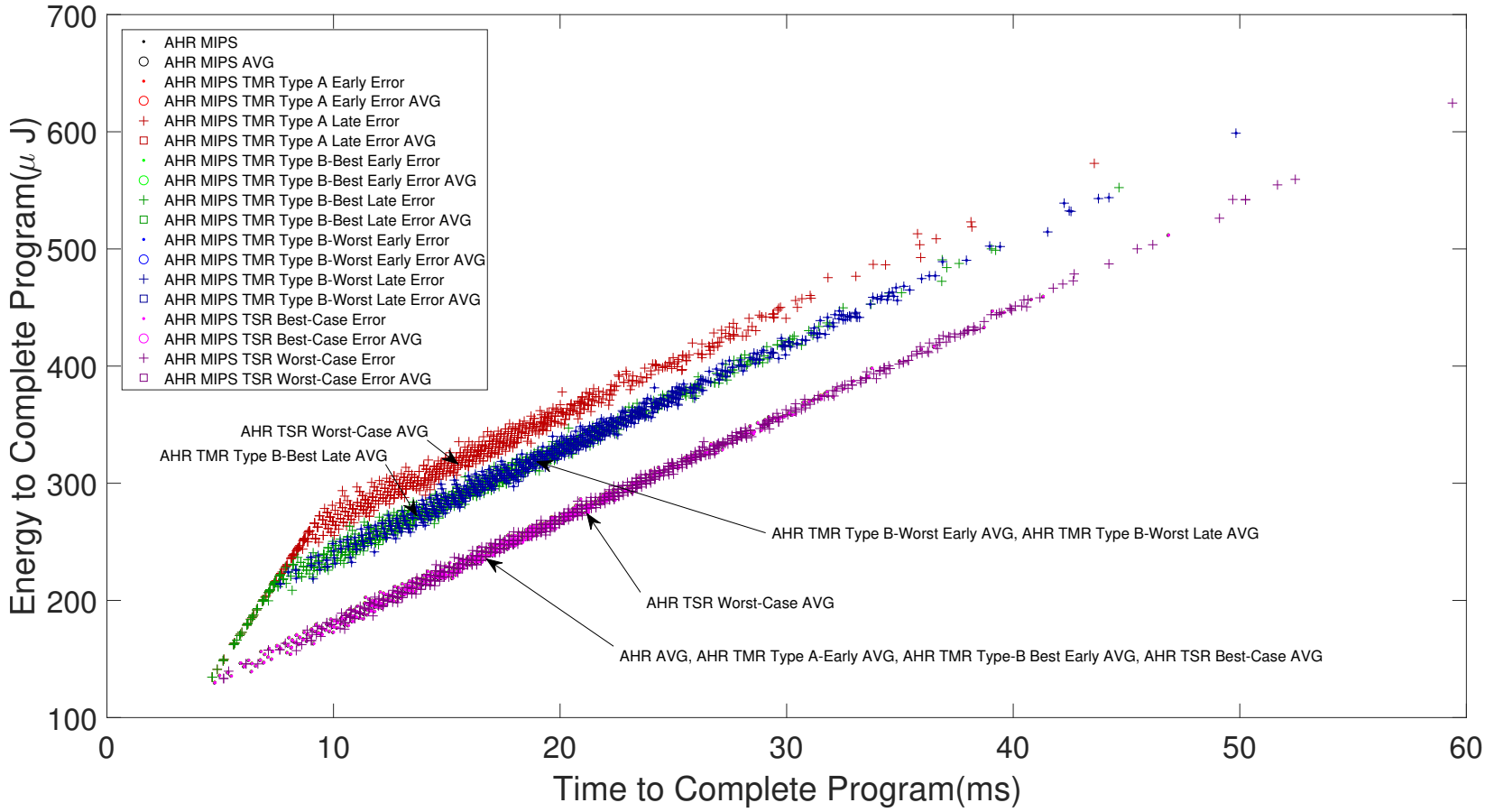

Figure 52. Software Simulation of AHR MIPS Errors - Energy vs. Time to Complete

Figure 53 shows the average time and energy for TMR MIPS, TSR MIPS, and AHR MIPS to complete programs in the presence and absence of errors. This figure illustrates how AHR MIPS bridges the gap between TMR MIPS and TSR MIPS performance. The AHR MIPS TMR Type A and Type B-Best errors appear to fall on a line between the TSR MIPS Best-case error and TMR MIPS Type A and Type B-Best errors. A similar pattern appears for AHR MIPS TMR Type B-Worst and TSR Worst-case errors which appear to nearly fall on a line between the TMR Type B-Worst and TSR Worst-case errors. However, this figure does not tell the entire story as the best-case, worst-case, early, and later errors define boundaries for AHR MIPS performance in the presence of errors.

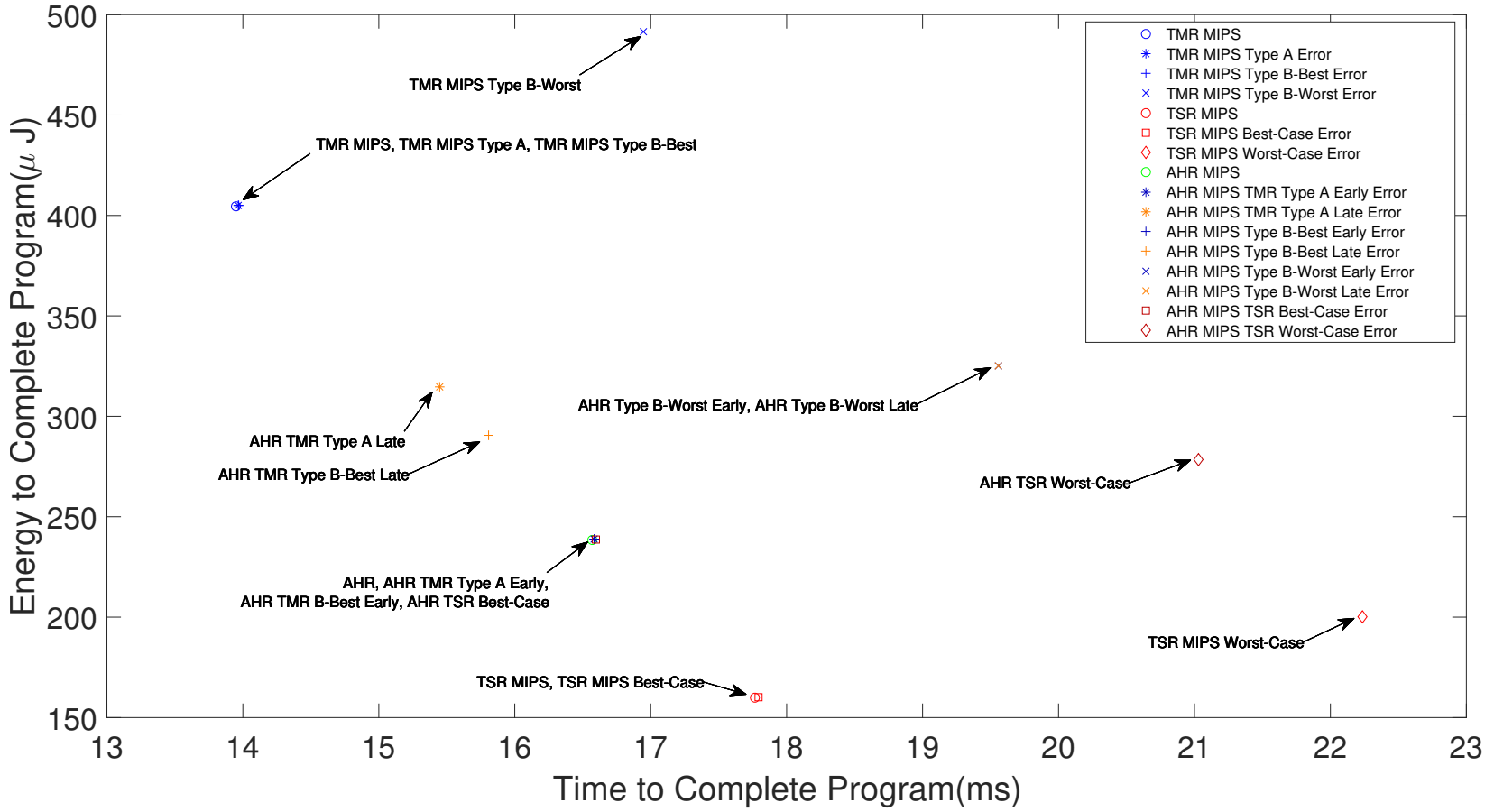

Figure 53. Averaged Results of Software Simulation of All Errors - Energy vs. Time to Complete

Figure 54 shows the bounding boxes when the TMR to TSR transition point occurs at 15,000 instructions. Note that the points plotted in this figure are the same as those plotted in Figure 53 and represent the average program completion times; however, the TMR Type B-Best, TSR Best-Case, AHR TMR Type A Early, AHR TMR Type B-Best Early, and AHR TSR Best-Case errors have been omitted from this plot. The TMR Type B-Best case error result was nearly identical to the TMR MIPS with no error result. The TSR MIPS Best-Case error result was nearly identical to the TSR MIPS with no error result. The AHR TMR Type A Early, AHR TMR Type B-Best Early, and AHR TSR Best-Case error results were nearly identical to the AHR MIPS with no error result. The bounding boxes indicate that the average program completion time should fall somewhere within the bounding box when errors are present.

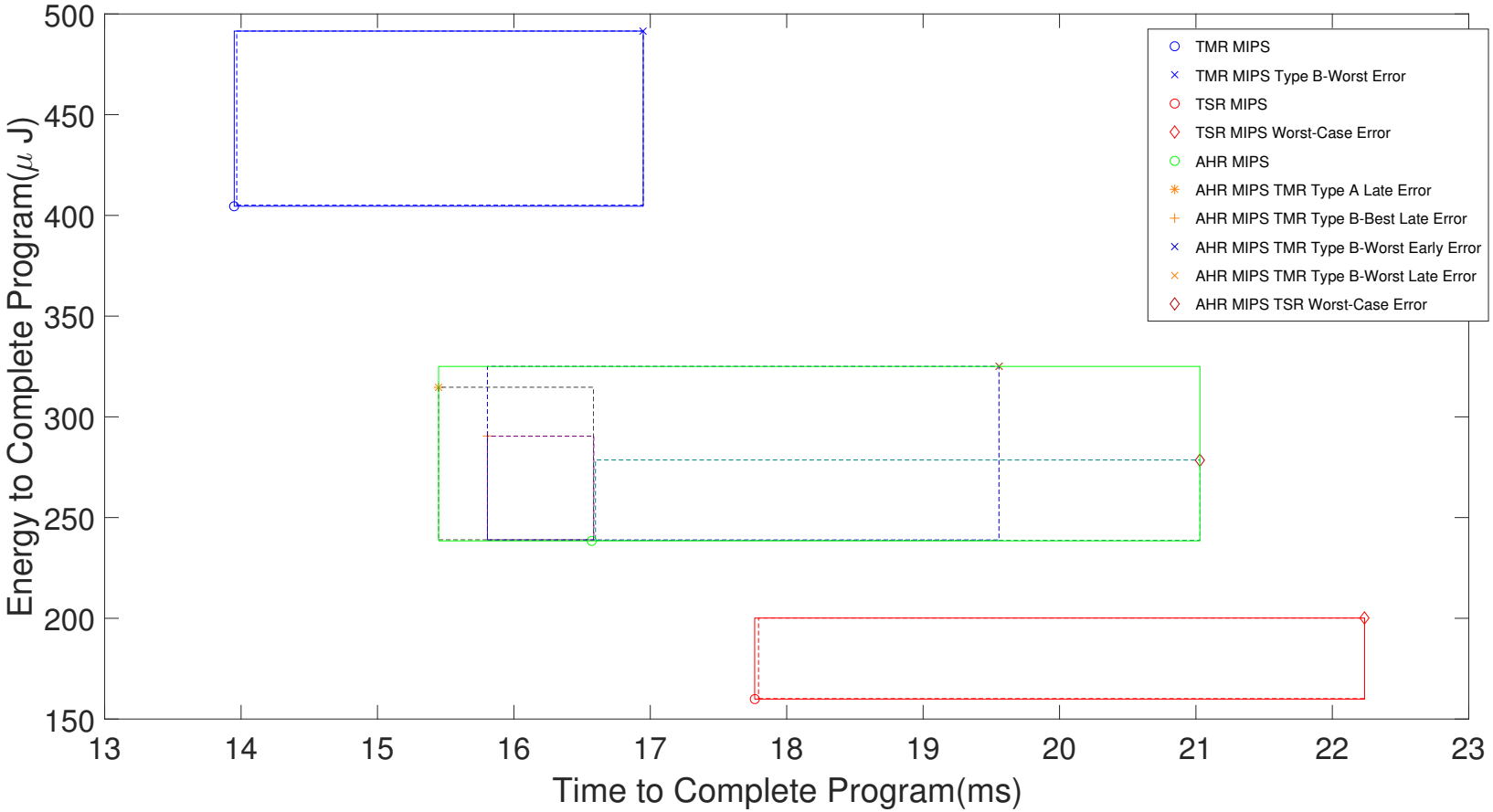

Figure 54. Average Performance Bounds for AHR MIPS with a TMR to TSR Point at 15,000 Instructions

The corners of the bounding box encompassing the TMR MIPS Type B-Best and Type B-Worst errors is shown as a dotted blue line. This box indicates that average program completion time and energy usage for a TMR MIPS program encountering a Type B error will end up within this box and it nearly overlaps the second box indicated by a solid blue line. The second box is used to outline the average performance of TMR MIPS when errors may or may not be present. It includes the no error, TMR Type A, and TMR Type B errors.

The corners of the bounding box encompassing the TSR MIPS Best- and Worst-case errors is shown as a red dashed line with the red square and the red diamond at opposing corners to indicate the average Best- and Worst-case error program runtime and energy usage. A second box with a red solid line is used to outline the average performance of TSR MIPS when errors may or may not be present. It includes the no error, TSR Best-case, and TSR Worst-case errors. Once again, these boxes almost overlap one another.

The corners of the bounding box encompassing the AHR MIPS TMR Type A Early and Late errors is shown as a gray dashed line to indicate how a program will perform in terms of time and energy usage on average when a TMR Type A error will occur. Similarly, the purple dashed line indicates the average performance of a program experiencing a TMR Type B-Best case error. The orange dashed line indicates the average performance of a program experiencing a TMR Type B-Worst error, however, no such box is visible in this figure as the TMR Type B-Worst Early and Late errors are identical in this figure. A dark blue dashed line bounding box extends from the left most plus sign (+) to the right most "X" and from the AHR MIPS no error (green circle) to the top most "X" to show the average bounds of a AHR MIPS program that encounters any TMR Type B error. The dashed teal line indicates the bounds of a AHR MIPS TSR error. Finally, the solid green line

indicates the average bounds for a AHR MIPS program experiencing any TMR error, TSR error, or no error.

Note that the portion of the bounding box extending to the left of the average AHR MIPS no error runtime and energy usage does not necessarily indicate that an error could occur such that the runtime would decrease without a change in energy usage. It should be expected that a decrease in runtime would correspond to a greater number of instructions being performed in TMR mode and a resulting energy increase; however, there is insufficient analysis at this time to determine a more precise boundary region and the creation of such a region is left for future work.

Now, because the bounding boxes indicate that the average time and energy used to complete a program in the presence of errors should fall within the boxes, they should not be treated as program specific bounding boxes. It would be trivial to create program specific bounding boxes, but these are not shown here for brevity. However, the next figures will begin to show the versatility of the AHR MIPS approach as the TMR to TSR transition point is varied.

Figures 55 to 62 show what happens to the bounding boxes as the TMR to TSR transition point increases from 11,000, to 20,000, 30,000, 40,000, 50,000, 60,000, 70,000, and 80,000 instructions. As the transition point increases, The overall bounding box begins to grow, then shrinks down to match the TMR MIPS bounding box. Note that in some of these figures, the AHR MIPS TMR Type B-Worst Early and Late errors do not always coincide. Also note how the size and shapes of the smaller bounding boxes change. As the TMR to TSR transition point increases, the AHR MIPS TMR Type B-Best bounding box increases in size, then decreases in size until it becomes nonexistent. The AHR MIPS TMR Type B-Worst bounding box increases in size from nonexistence, then decreases in size until becoming nonexistent again. The AHR MIPS TMR Type A box also increases then decreases in size until becom-

ing nearly nonexistent. The AHR MIPS TSR box decreases in size until becoming nearly nonexistent. Figure 63 shows what it looks like when the bounding boxes from Figures 55 to 62 are overlaid on the same figure.

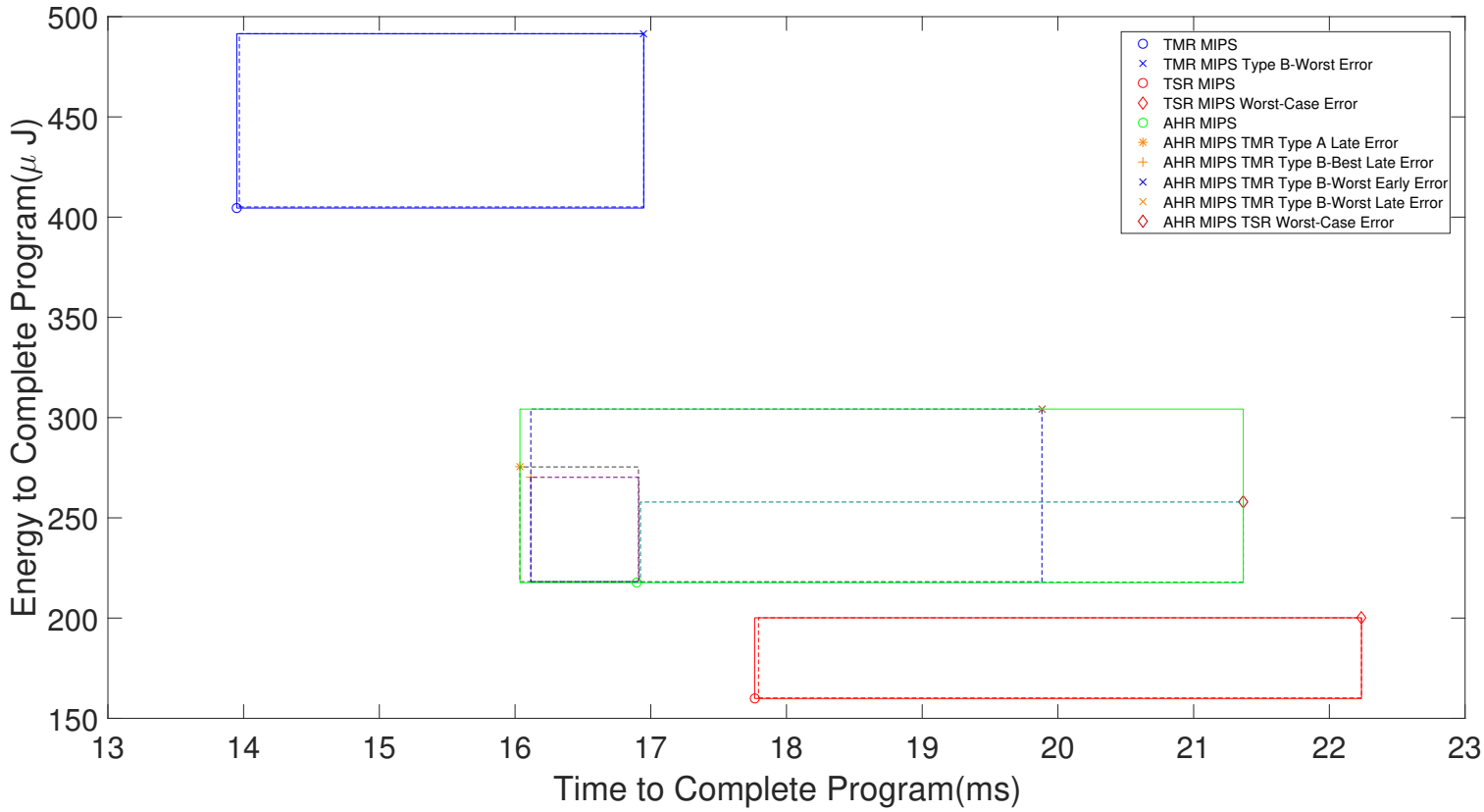

Figure 55. Average Performance Bounds for AHR MIPS with a TMR to TSR Point at 11,000 Instructions

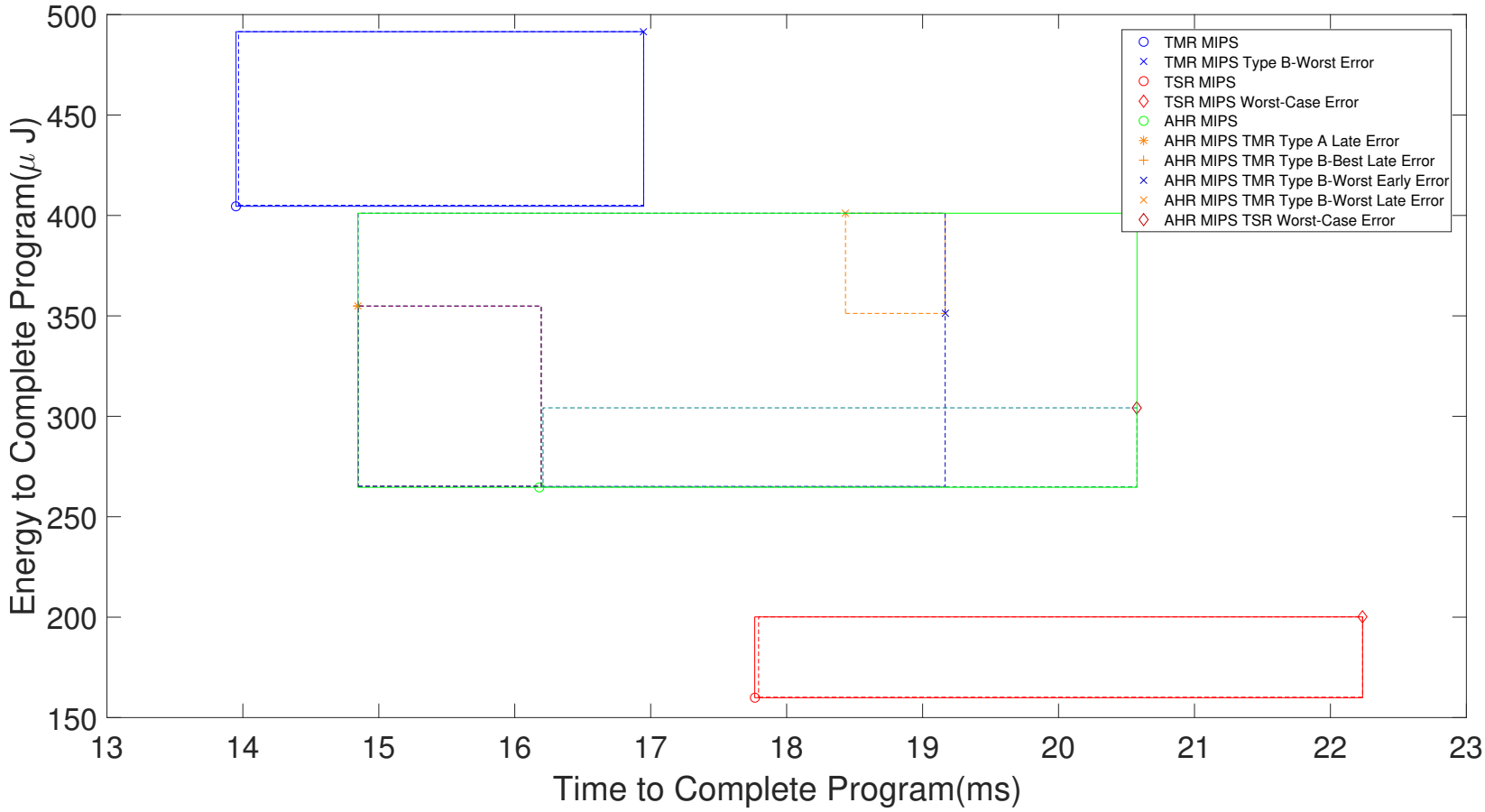

Figure 56. Average Performance Bounds for AHR MIPS with a TMR to TSR Point at 20,000 Instructions

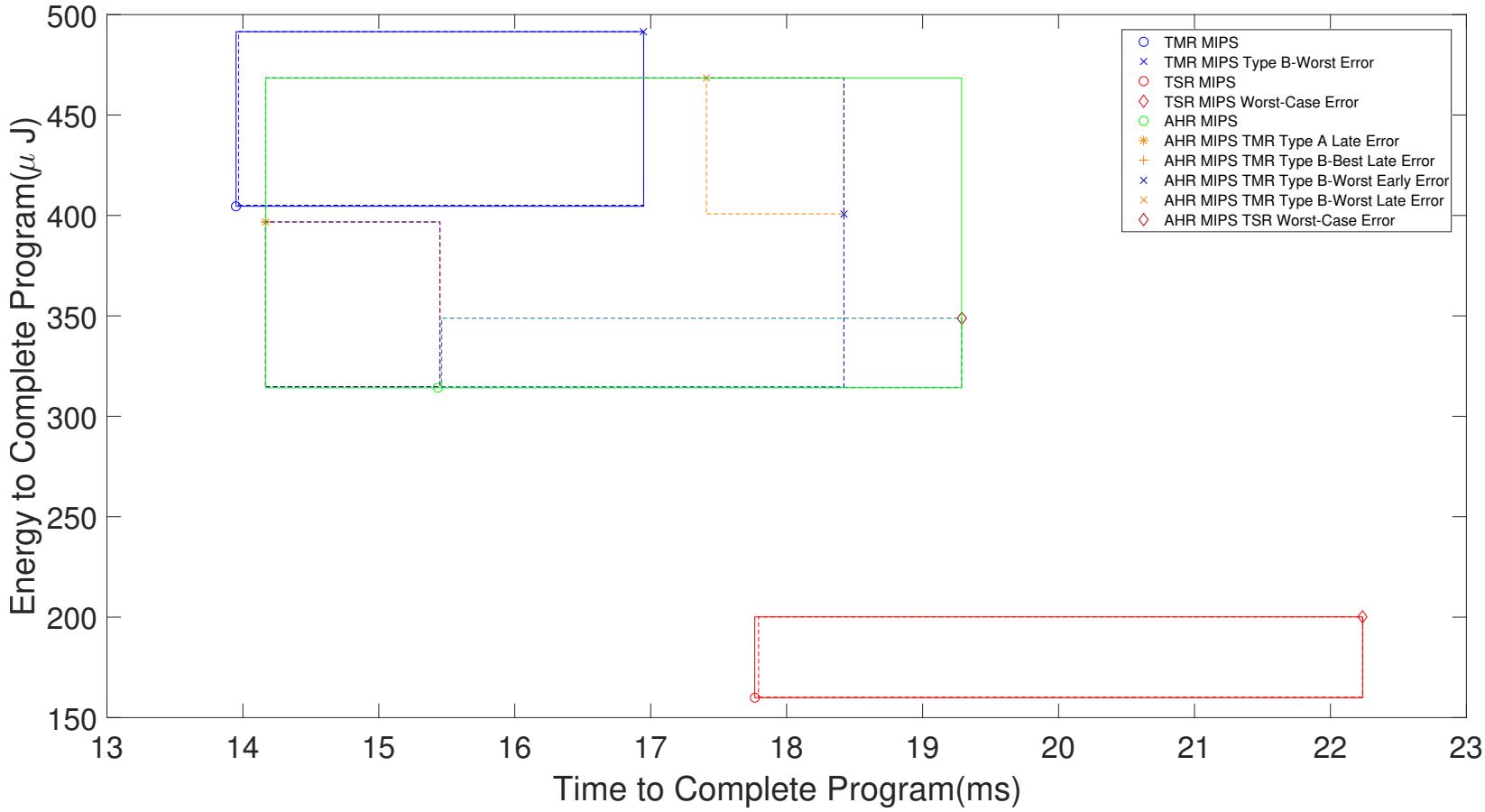

Figure 57. Average Performance Bounds for AHR MIPS with a TMR to TSR Point at 30,000 Instructions

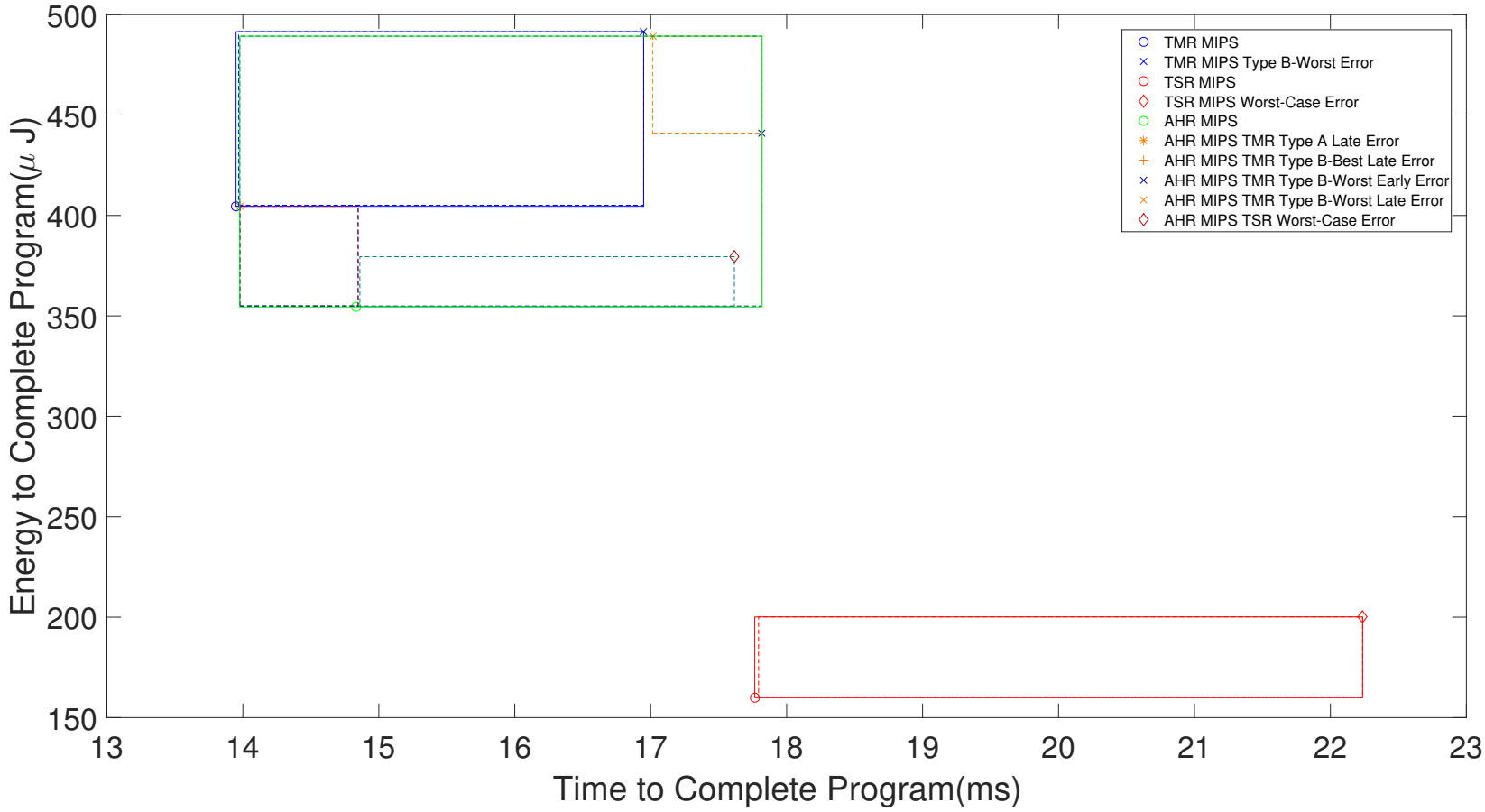

Figure 58. Average Performance Bounds for AHR MIPS with a TMR to TSR Point at 40,000 Instructions

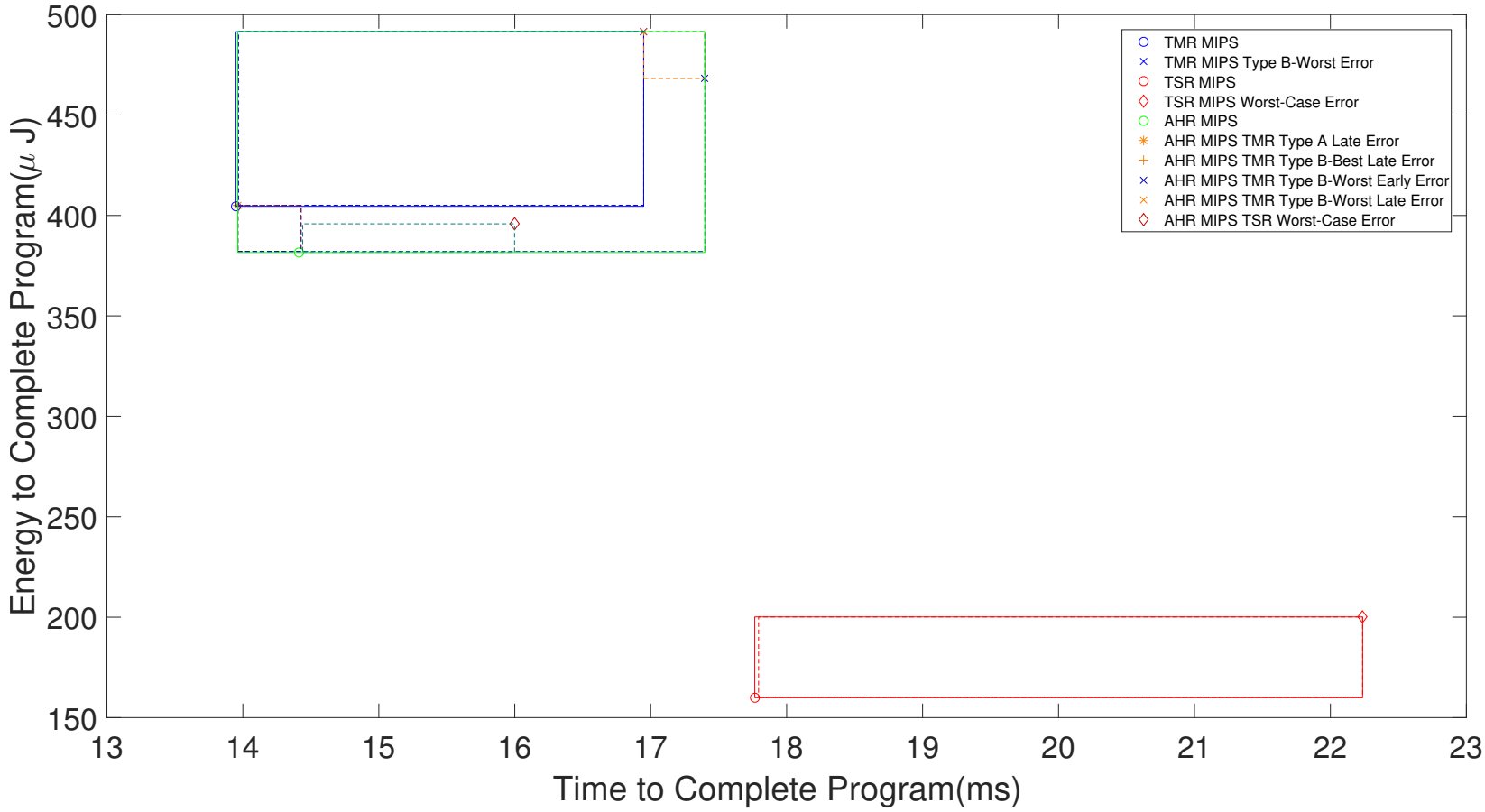

Figure 59. Average Performance Bounds for AHR MIPS with a TMR to TSR Point at 50,000 Instructions

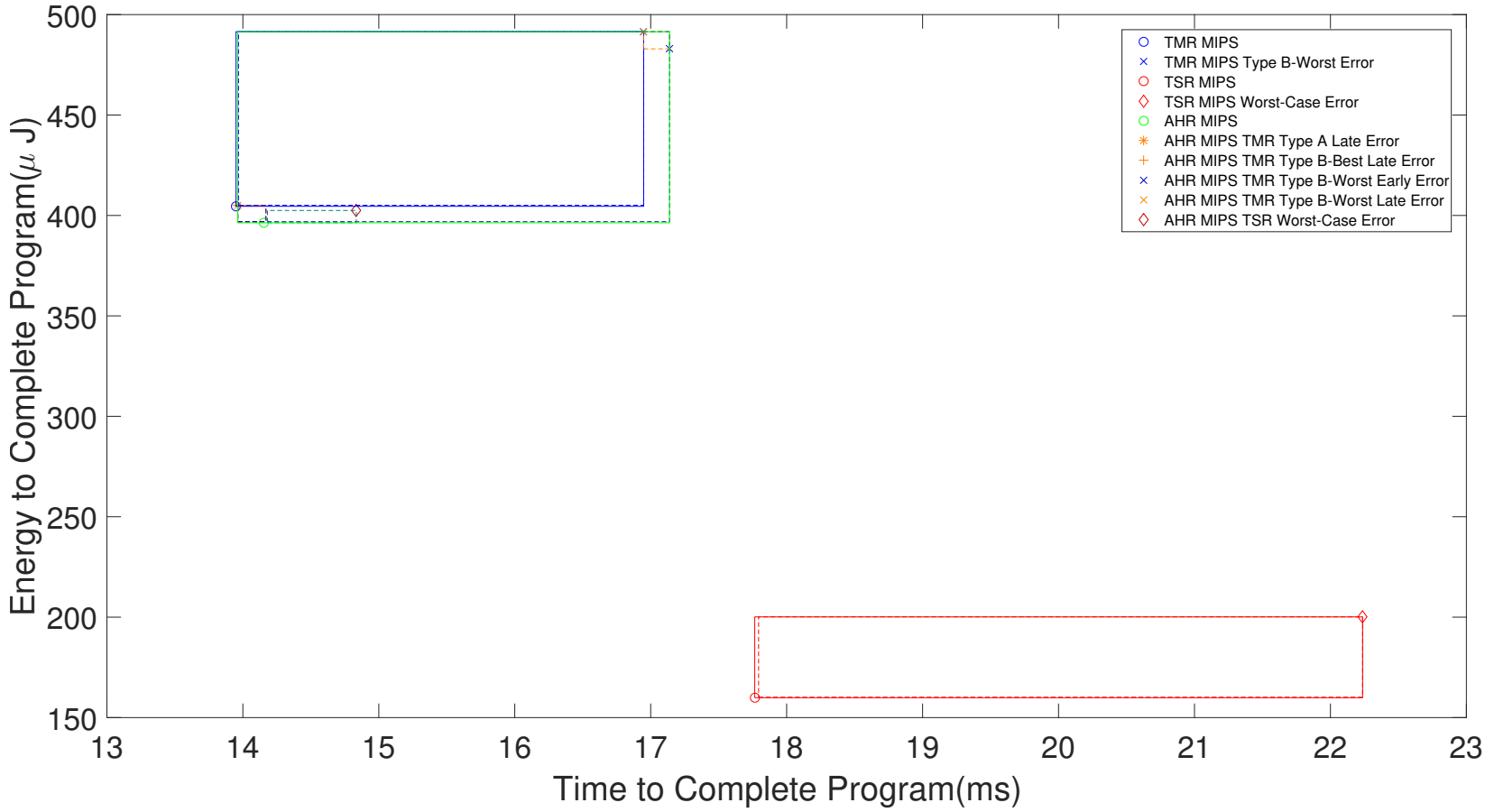

Figure 60. Average Performance Bounds for AHR MIPS with a TMR to TSR Point at 60,000 Instructions

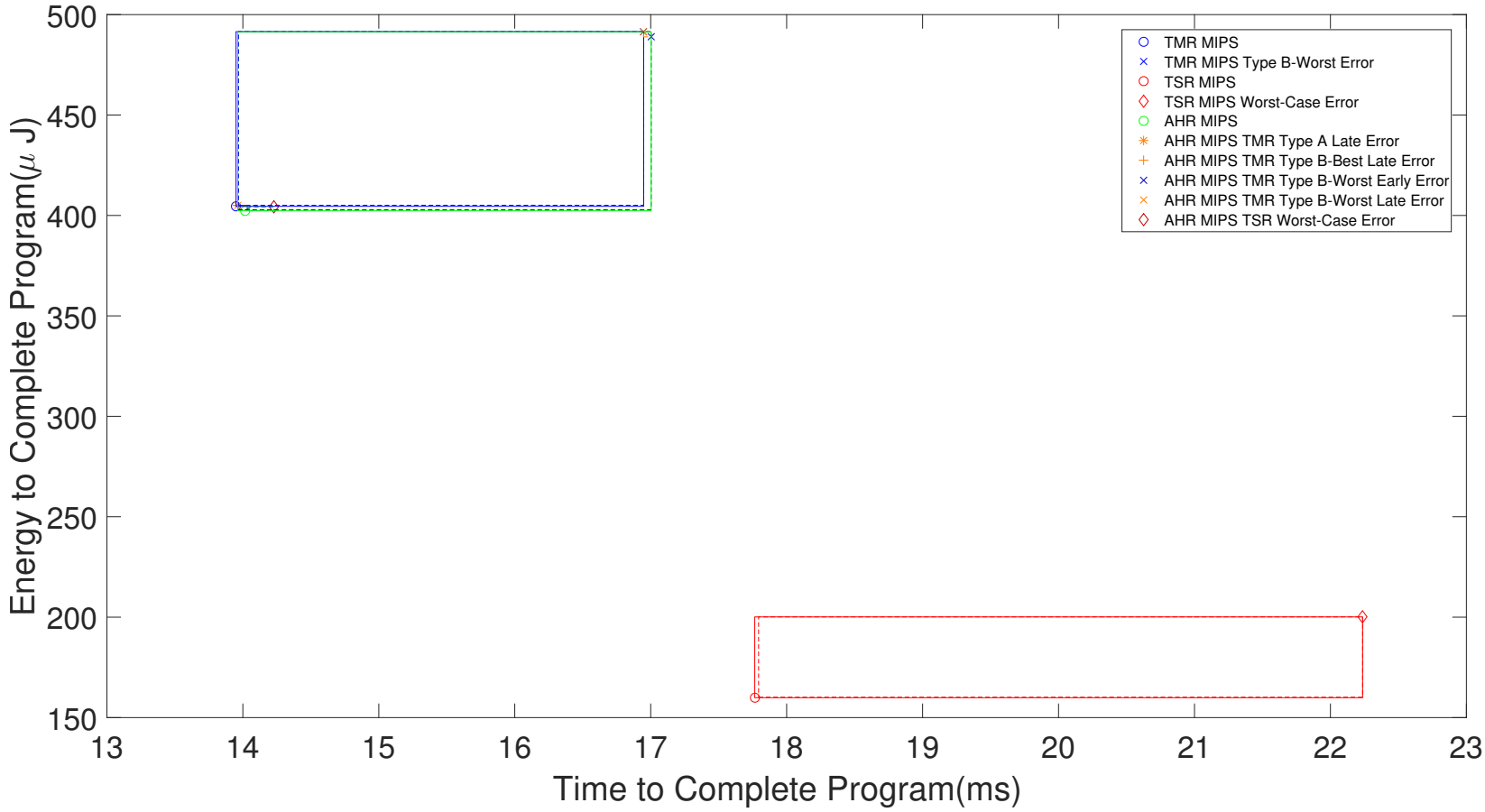

Figure 61. Average Performance Bounds for AHR MIPS with a TMR to TSR Point at 70,000 Instructions

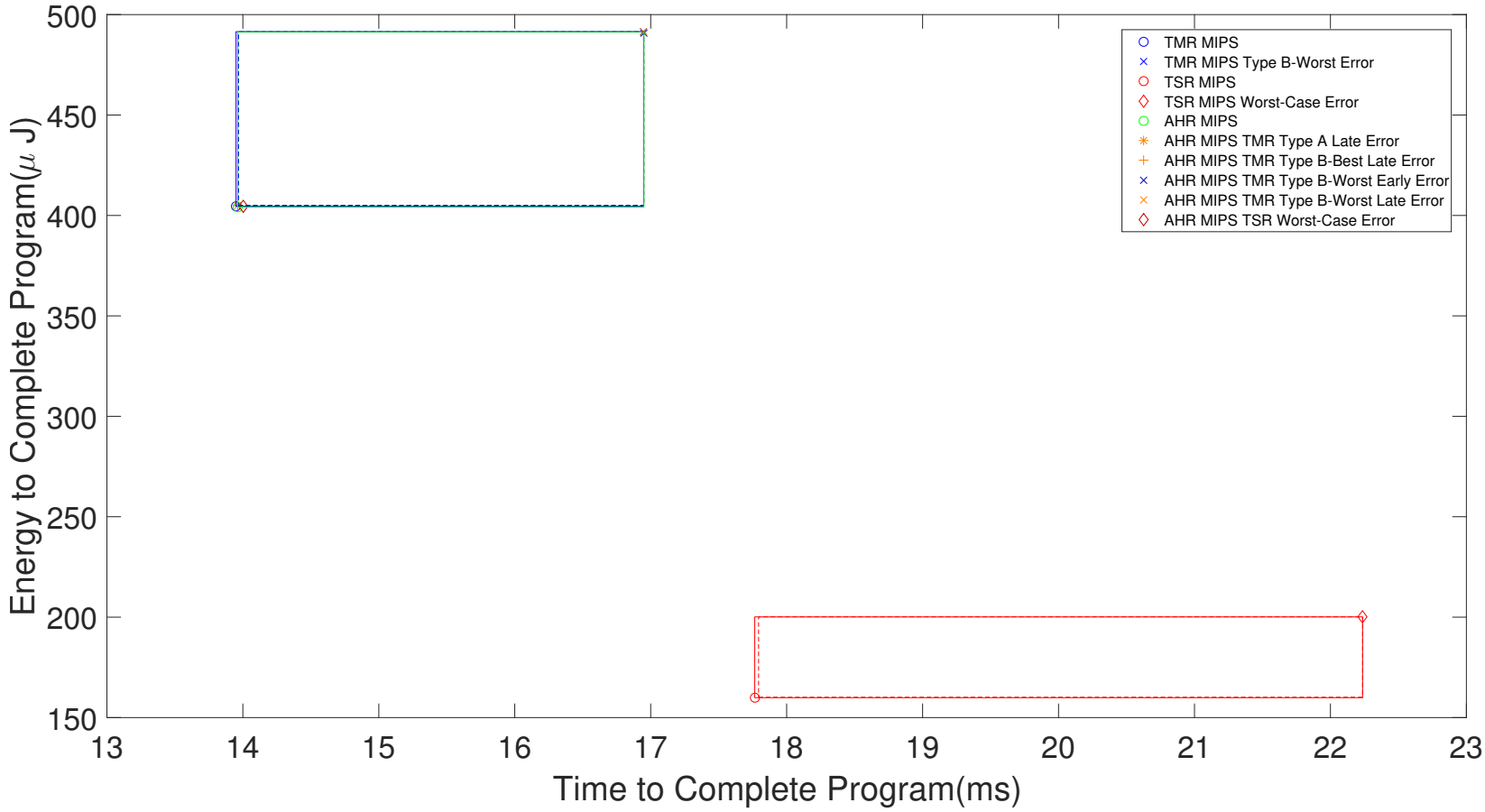

Figure 62. Average Performance Bounds for AHR MIPS with a TMR to TSR Point at 80,000 Instructions

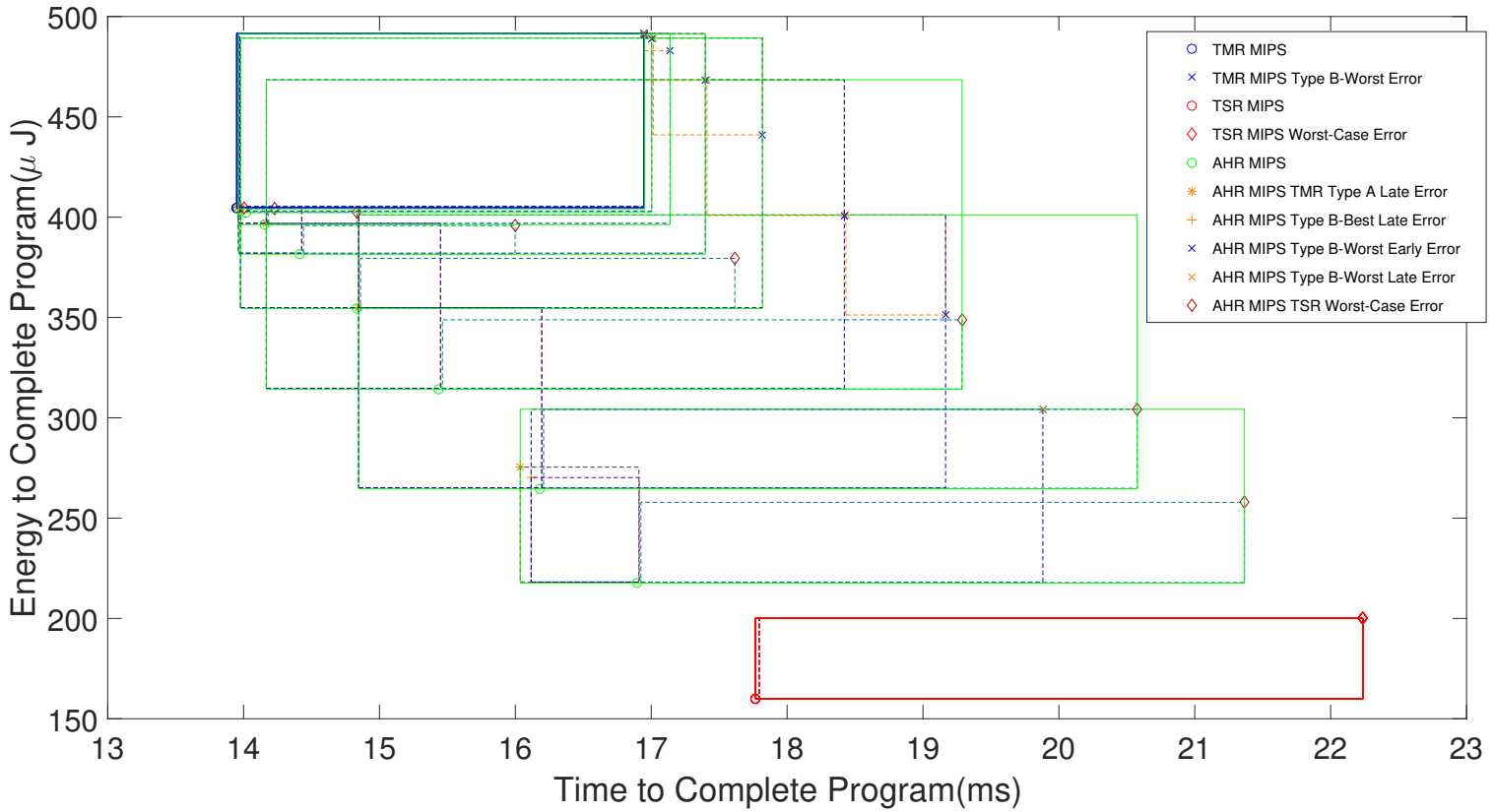

Figure 63. Average Performance Bounds for AHR MIPS with a TMR to TSR Point varying from 11,000 to 80,000 Instructions

While these figures represent the average performance for 1,000 programs, they have greater utility when created for a specific program to show how the expected program runtime and energy usage change as the TMR to TSR transition point is changed. A satellite designer, mission planner, or operator could use these to determine the best transition point based on the needs of the system. For example, the TMR to TSR transition point could be selected in order to meet certain performance criteria such as staying under maximum runtime or energy constraints.

Figure 64 shows the same things as Figure 53, but allows the TMR to TSR transition point to vary from 11,000 to 80,000 instructions in increments of 1,000. This figure also provides a slightly different view to the bounding boxes in the previous figures. It is most useful in visualizing how the average program runtime and energy usage for each error scenario changes as the TMR to TSR transition point changes. Curves for all AHR MIPS error scenarios, and the no error scenario, become evident. When there are only 11,000 instructions completed in TMR before the TMR to TSR transition point, AHR MIPS behaves much more closely to TSR MIPS. As the transition point moves towards 80,000 instructions, the AHR MIPS results begin moving up and to the left until they coincide with the TMR MIPS results. Note that the AHR MIPS TSR Best- and Worst-case scenarios collapse to the no error solution for TMR MIPS when very little, if any time is spent in TSR MIPS because the TMR to TSR transition point is no longer reached during the duration of most programs. Similarly, the AHR MIPS TMR Type B-Worst scenarios converge to the TMR Type B-Worst error scenario.

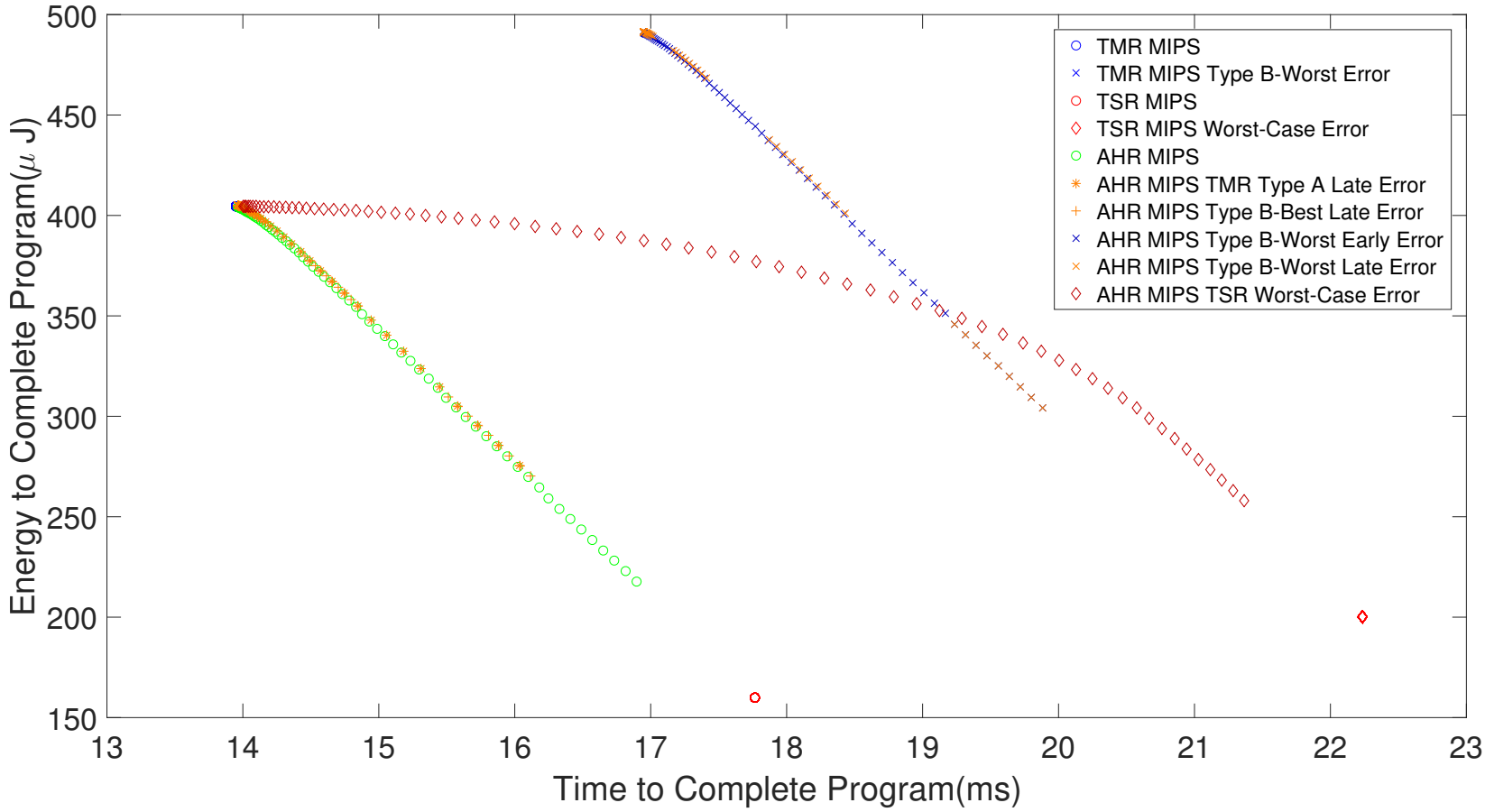

Figure 64. TMR to TSR Transition Varying from 11,000 to 80,000 Instructions - Energy vs. Time to Complete

Another interesting comparison is to look at the average percent difference in runtime and energy usage for each error scenario and no error scenario when compared to Basic MIPS with no errors. The average percent difference for the no error scenarios were previously given in Equations 17 to 22. The average percent difference for the programs experiencing errors are given in Equations 88 to 113.

$$PD_{Time\ TMR\ ErrA\ v\ Basic} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{TMR\ ErrA}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{88}$$

$$PD_{Time\ TMR\ ErrB\ Best\ v\ Basic} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{TMR\ ErrB\ Best}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{89}$$

$$PD_{Time\ TMR\ ErrB\ Worst\ v\ Basic} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{TMR\ ErrB\ Worst}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{90}$$

$$PD_{Time\ TSR\ Best\ v\ Basic} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{TSR\ Best}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{91}$$

$$PD_{Time\ TSR\ Worst\ v\ Basic} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{TSR\ Worst}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{92}$$

$$PD_{Time\ CTMR\ A\ Early\ v\ Basic} = \cdots \frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{CTMR\ A\ Early}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{93}$$

$$PD_{Time\ CTMR\ A\ Late\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{CTMR\ A\ Late}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{94}$$

$$PD_{Time\ CTMR\ B\ Best\ Early\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{CTMR\ B\ Best\ Early}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{95}$$

$$PD_{Time\ CTMR\ B\ Best\ Late\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{CTMR\ B\ Best\ Late}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{96}$$

$$PD_{Time\ CTMR\ B\ Worst\ Early\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{CTMR\ B\ Worst\ Early}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{97}$$

$$PD_{Time\ CTMR\ B\ Worst\ Late\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{CTMR\ B\ Worst\ Late}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{98}$$

$$PD_{Time\ CTSR\ Best\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{CTSR\ Best}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{99}$$

$$PD_{Time\ CTSR\ Worst\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{T_{CTMR\ Worst}(n) - T_{Basic\ MIPS}(n)}{T_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{100}$$

$$PD_{Energy\ TMR\ ErrA\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{TMR\ ErrA}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{101}$$

$$PD_{Energy\ TMR\ ErrB\ Best\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{TMR\ ErrB\ Best}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{102}$$

$$PD_{Energy\ TMR\ ErrB\ Worst\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{TMR\ ErrB\ Worst}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{103}$$

$$PD_{Energy\ TSR\ Best\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{TSR\ Best}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{104}$$

$$PD_{Energy\ TSR\ Worst\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{TSR\ Worst}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{105}$$

$$PD_{Energy\ CTMR\ A\ Early\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{CTMR\ A\ Early}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{106}$$

$$PD_{Energy\ CTMR\ A\ Late\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{CTMR\ A\ Late}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{107}$$

$$PD_{Energy\ CTMR\ B\ Best\ Early\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{CTMR\ B\ Best\ Early}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{108}$$

$$PD_{Energy\ CTMR\ B\ Best\ Late\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{CTMR\ B\ Best\ Late}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{109}$$

$$PD_{Energy\ CTMR\ B\ Worst\ Early\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{CTMR\ B\ Worst\ Early}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{110}$$

$$PD_{Energy\ CTMR\ B\ Worst\ Late\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{CTMR\ B\ Worst\ Late}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{111}$$

$$PD_{Energy\ CTSR\ Best\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{CTSR\ Best}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{112}$$

$$PD_{Energy\ CTSR\ Worst\ v\ Basic} = \cdots$$
$$\frac{\sum_{n=1}^{N_{programs}} \left[\frac{E_{CTSR\ Worst}(n) - E_{Basic\ MIPS}(n)}{E_{Basic\ MIPS}(n)} \times 100\% \right]}{N_{programs}} \tag{113}$$

The average percent difference equations were used to calculate the average per-

cent difference for all programs experiencing errors and no errors when the TMR to TSR transition point from 11,000 to 80,000 in increments of 1,000. The results for the runtime calculations are shown in Figure 65 and the results for energy usage calculations are shown in Figure 66. These figures really highlight how AHR MIPS runtime and energy performance changes when compared to Basic MIPS as the TMR to TSR transition point changes. As in some of the previous figures, the TMR Type A and TMR Type B-Best error results are omitted because they are nearly identical to the TMR no error results. The same is true for the TSR Best error results because they are identical to the TSR no error results. Additionally, the AHR TMR Type A Early, AHR TMR Type B-Best Early, and AHR TSR Best error results have been omitted because they are nearly identical to the AHR no error results. The first thing to note from these figures is how the AHR MIPS no error, AHR TMR Type A, AHR TMR Type B-Best, AHR TSR Best, and AHR TSR Worst average percent differences approach the TMR average percent difference as the number of instructions before the TMR to TSR transition increases. This is consistent with prior results because AHR MIPS performance is nearly identical to TMR MIPS performance as the number of instructions that AHR MIPS processes in TSR mode approaches zero and nearly all instructions are processed in TMR mode. Additionally, the AHR MIPS TMR Type B-Worst average percent differences approach the TMR Type B-Worst average percent difference, which is also expected for the same reasons just given.

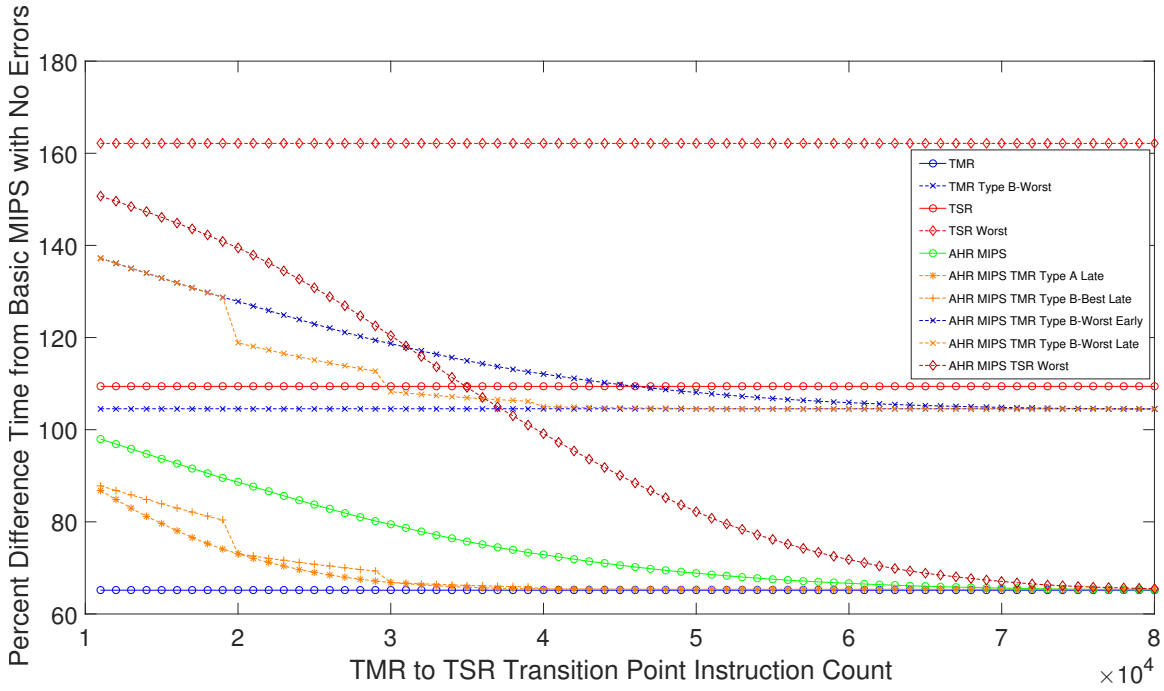

Figure 65. AHR MIPS TMR to TSR Transition Varying from 11,000 to 80,000 Instructions - Energy vs. Time to Complete

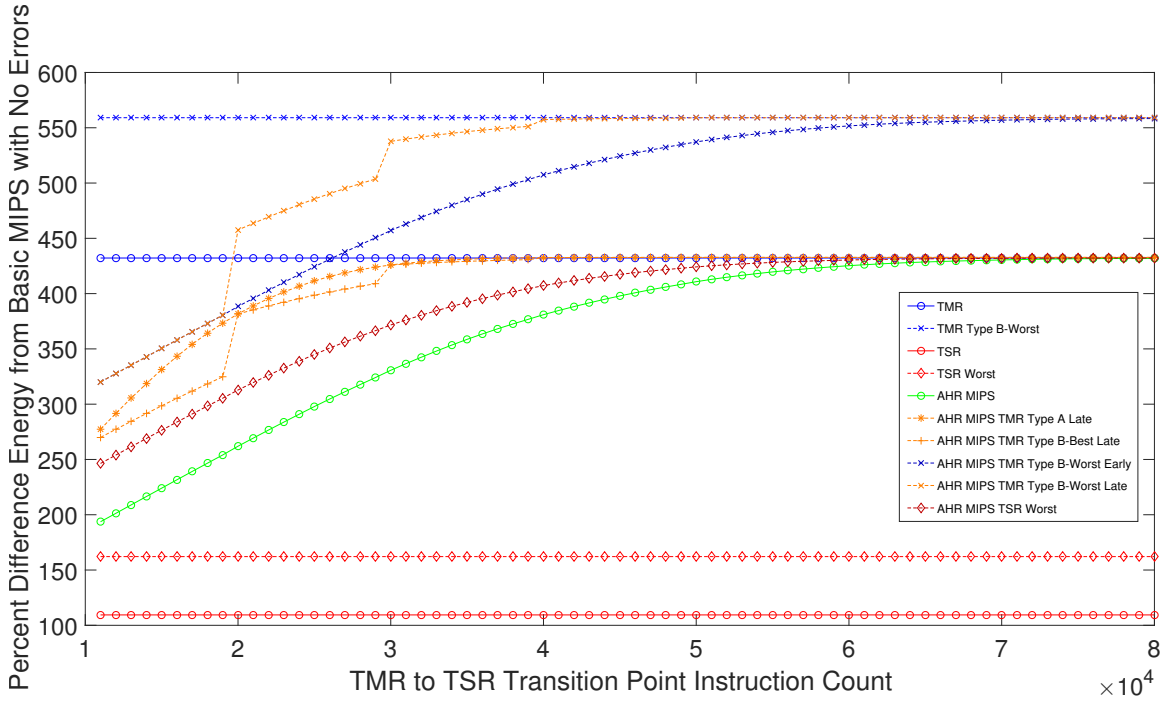

Figure 66. AHR MIPS TMR to TSR Transition Varying from 11,000 to 80,000 Instructions - Energy vs. Time to Complete

There are a few other things to note from the percent difference figures. The first is that programs experiencing an AHR TMR Type A Late error complete faster than programs experiencing a AHR TMR Type B-Best Late error. Both of these complete faster than AHR MIPS programs experiencing no error, AHR TMR Type A Early, AHR TMR Type B-Best Early, and AHR TSR Best-case errors. The no error, AHR TMR Type A Early, AHR TMR Type B-Best Early, and AHR TSR Best-case error scenarios all take less time to complete than programs experiencing AHR TMR Type B-Worst Early, AHR TMR Type B Worst Late, and AHR TSR Worst-case errors. Programs experiencing AHR TMR Type B-Worst Late errors always complete faster than those experiencing AHR TMR Type B-Worst Early errors. AHR MIPS programs experiencing TSR Worst-case errors have the worst runtime when the TMR to TSR transition point is under about 30,000, but runs faster than programs experiencing TMR Type B-Worst Early errors when the transition point is greater than 31,000 instructions and faster than programs experiencing TMR Type B-Worst Late errors when the transition point is greater than about 37,000 instructions.

AHR MIPS programs experiencing no error, TMR Type A Early, TMR Type B-Best Early, and TSR Best-case all take about the same amount of energy to complete and use less energy than a AHR MIPS program experiencing any other type of error. AHR MIPS programs experiencing TMR Type B-Worst Late errors use the most energy followed by programs experiencing TMR Type B-Worst Early errors, then TMR Type A Late errors, then TSR Worst-case errors.

One final thing to note are the jump discontinuities in the AHR MIPS TMR Type B-Best Late and AHR MIPS TMR Type B-Worst Late timing and energy percent differences. These are a direct result of the TMR to TSR transition point moving passed one of the TMR save/restore point creation times which occur every 10,000 instructions. Note that these discontinuities occur as the TMR to TSR transition

point passes 20,000, 30,000, and 40,000 instructions. This is because these late errors go from having a minimal impact when the TMR to TSR transition point occurs immediately before a save/restore point creation to a maximum impact when the TMR to TSR transition point occurs immediately after a save/restore point creation.

Figures 67 and 68 are essentially derivative plots of Figures 65 and 66 except that they use the average time and energy results rather than the percent differences. Each point on these graphs represent the difference in average time and average energy to complete 1,000 different programs with the given error type (or no error at all) from one AHR transition point value to the previous AHR transition point value where these transition points started at 11,000 instructions, ended at 80,000 instructions, and had step sizes of 1,000 instructions. Plots like these may help a mission planner determine the most optimal point, in terms of processing speed and energy usage, at which to transition AHR from TMR mode to TSR mode.

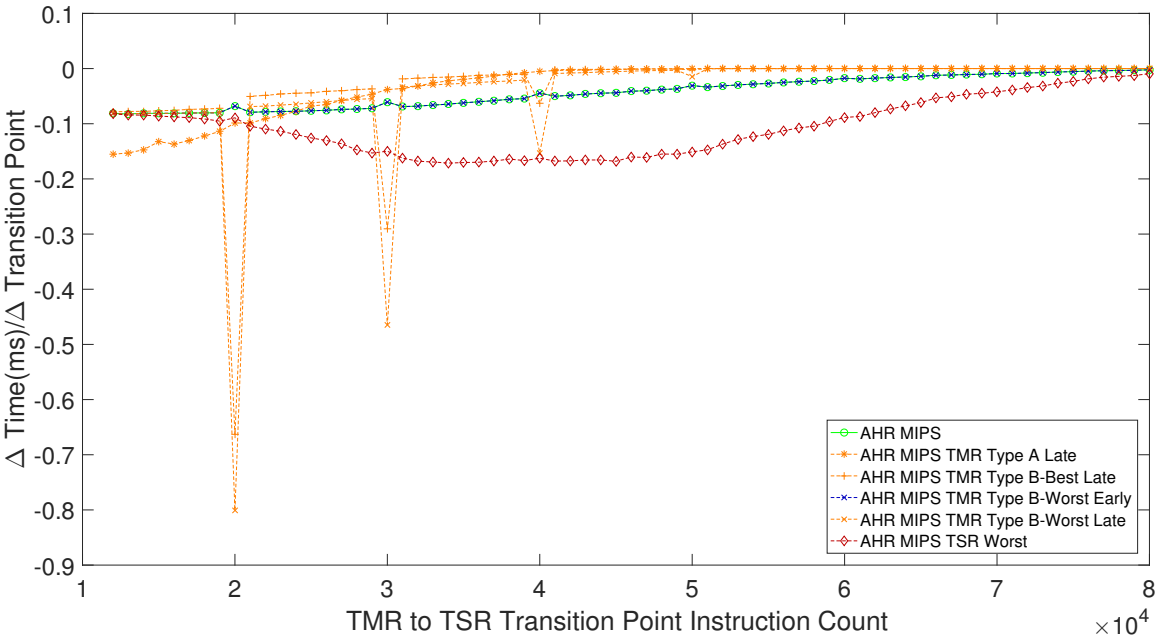

Figure 67. Time Difference Between Successive Steps of TMR to TSR Transition Point When Varying from 11,000 to 80,000 in Steps of 1,000

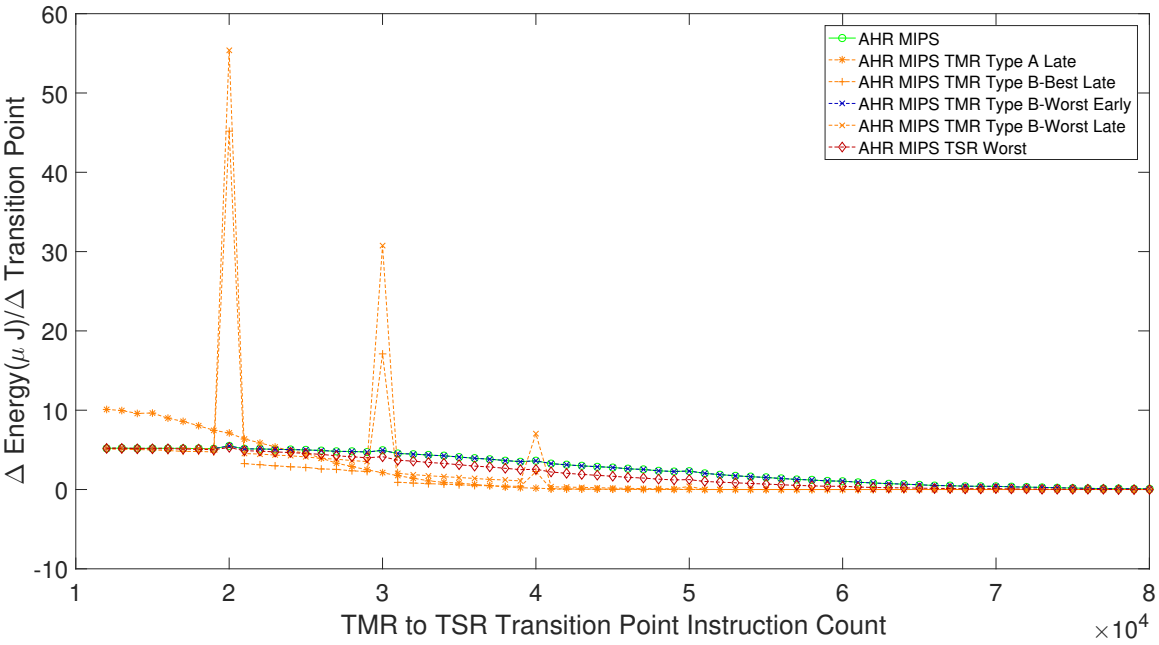

Figure 68. Energy Difference Between Successive Steps of TMR to TSR Transition Point When Varying from 11,000 to 80,000 in Steps of 1,000

6.3 HITL Simulation with Error Injection

Due to the difficulties described in Sections 4.5.2 and 4.5.4, only a few HITL simulations with error injection were performed, but these only examined error injection into TMR and TSR MIPS when each architecture was implemented on the same FPGA as its associated memulator. This means that timing measurements were recorded, but not energy measurements. AHR MIPS HITL simulations could not be accomplished because the combined size of AHR MIPS and the AHR memulator on the FPGA was such that timing did not close. This means that some of the signal wires on the FPGA were so long that it takes more than one clock cycle for them to travel from source to destination. The AHR MIPS HITL timing simulations should be completed in the future once the timing hurdle is overcome. Additionally, the energy measurements should be completed once the hardware problems associated with placing the processor and memory on separate boards have been resolved.

Table 13 shows the results of error injection for a single program implemented in TMR MIPS and TSR MIPS. Timing information was collected for each error type. The first column indicates the architecture and type of error (or no error) injected. The second column shows the time to complete the program according to simulation and analysis. The third column shows the measured time to complete the program in HITL simulations. The time to complete each program in this column is the average time to complete the program based upon ten runs of the program. The fourth column shows the time difference between the HITL simulation and the software simulation and analysis. The fifth column provides the same time difference information as the fourth column, but presents it in terms of 50MHz clock cycles rather than time in nanoseconds. All times shown are in nanoseconds.

Table 13. TMR MIPS and TSR MIPS HITL Timing Results With Error Injection for One Program

Architecture & Error Type	S&A	HITL	Difference	Clock Cycles
Basic MIPS No Err	8012480	8012566	86	4.30
TMR MIPS No Err	13254940	13255117	177	8.85
TMR MIPS Type A	13269740	13269436	-304	-15.20
TMR MIPS Type B-Best	13280080	13280266	186	9.30
TMR MIPS Type B-Worst	16295000	16283375	-11625	-581.25
TSR MIPS No Err	1628920	16299055	135	6.75
TSR MIPS Best	16334480	16328236	-6244	-312.20
TSR MIPS Worst	20403300	20399340	-3960	-198.00

Most of the HITL simulation results match the expected software simulation and analysis simulation results. Minor discrepancies are explained by clock jitter and skew in the FPGAs clock oscillator. In other words, the FPGA's clock oscillator is not able to provide a perfect 50MHz clock. Larger difference between HITL simulations and software simulations and analyses were noted for TMR MIPS Type B-Worst, TSR

MIPS Best, and TSR MIPS Worst errors. There are two possible explanations for these discrepancies. The first is that timing delays could have been introduced into the FPGA by the software that programs the FPGA. This could be due to signal routing timing issues and signal routing timing issues were observed during the first attempt at error free HITL simulation. The second is that there is some difficulty in translating the timing of error injection from software simulation and analysis to HITL simulations. This is because the software simulation and analysis uses one method of tracking the program loop count and processor program counter and the VHDL code uses a different method of tracking the program loop count and processor program counter. In software simulation, the loop count starts at one and increases to its final value while in VHDL code, the loop count starts at a high value and decreases to zero. Additionally, the HITL simulations track the program counter as starting at one and increasing by one at a time while the VHDL code tracks the program counter as starting at zero and increasing by four at a time. More work is needed to determine the cause of these large discrepancies.

6.4 Results Summary

This chapter discussed the results of the simulations and analyses that were proposed in Chapter V, Sections 5.4 and 5.5. The results demonstrated that Adaptive-Hybrid Redundancy (AHR) provides an advantage over standalone Triple Modular Redundancy (TMR) or Temporal Software Redundancy (TSR). Regardless of whether errors are present, AHR uses less energy than TMR and processes faster than TSR. However, it does this at the expense of not being as fast as TMR or as energy efficient as TSR. However, with the AHR approach, a satellite designer now has the ability to choose when to transition from TMR to TSR based on the performance needs of the mission.

VII. Conclusions

The conclusion of this dissertation returns to the research question posed at the beginning of this dissertation.

1. Can multiple redundancy methods be incorporated into the redundancy design?
2. Is it possible to allow flexibility in redundancy methods for the duration of a space vehicle's lifetime?
3. Is it possible to switch between these methods based on mission needs?
4. What are the timing and energy tradespaces available to a designer, mission planner, or operator?

To answer the first question, an Adaptive-Hybrid Redundancy (AHR) architecture called AHR MIPS was developed in Chapter III and its function was verified in Section 4.2, thus demonstrating that at least two redundancy methods could be incorporated into a redundancy design.

The second, third, and fourth questions were answered in Chapters IV and VI. Chapter IV examined the performance of AHR in the absence of errors and demonstrated the ability of AHR to switch between TMR and TSR modes at any time which provides a "Yes" answer to the second and third questions. Chapter IV also explored the AHR time and energy tradespace in the absence of errors. This exploration demonstrated that the selection of the point at which AHR transitions from TMR to TSR has a direct impact on processing speed and energy usage. The more time AHR spends in TMR mode, the more closely AHR's performance mirrors TMR performance. Similarly, the more time AHR spends in TSR mode, the more closely AHR's performance mirrors TSR performance. Splitting the time between TMR and

TSR modes allows for speed and energy performance that neither TMR or TSR could achieve alone. However, Chapter IV only answers these questions in the absence of errors. VI answers these questions in the presence of errors. Once again, the answers to questions two and three are both "Yes" because AHR demonstrated the ability to switch between TMR and TSR even when errors occurred. VI also explored the AHR time and energy tradespace in the presence of errors. As AHR spends more time in TMR mode, the impact of TSR errors on time and energy performance are diminished. Similarly, as AHR spends more time in TSR mode, the impact of TMR errors on time and energy performance are diminished.

Adaptive-Hybrid Redundancy (AHR) has been proven by analysis and simulation to be a viable approach to combine disparate redundancy techniques in a single architecture to allow space systems designers and operators the flexibility and tradespace to maximize performance and energy efficiency during both the design phase and operational phase of a mission. This added flexibility also means that a particular mitigation strategy is not permanent once the space system has been designed and built. For previous systems, a single mitigation strategy was selected and could not be changed for the duration of the mission. Admittedly, many previous mitigation strategies are more complex than the simple TMR or TSR examined in this work, but none allowed significant changes to the mitigation strategy after implementation. The AHR approach provides the ability to operate fully in TMR, TSR, or anywhere in between depending on the setting of the TMR to TSR transition point at any time during the life of the mission.

In addition to showing AHR to be an excellent mitigation approach, this work also further explored the relative advantages and disadvantages of TMR and TSR, contrasted against an unmitigated strategy and confirmed many expectations.

For example, it was shown that TMR processes faster and uses more energy than

TSR as expected. It was also shown that a TMR mitigation strategy takes longer to complete programs than a no mitigation strategy due to the overhead of all memory interactions being checked by a voter in the TMR approach and the need to create save/restore points to periodically store the processors' internal state. It was also shown that TMR uses more than three times as much energy to complete programs than the unmitigated approach due to the triplication of processors, addition of the voter, and extra time needed to complete a program.

It was also shown that TSR takes at least twice as long as the unmitigated approach due to the duplication of instructions and need to create save/restore points to periodically store the processor's internal state. It was also shown that TSR uses more than double the energy to complete programs than the unmitigated approach due to the extra time required to complete TSR programs.

Finally, the Cyclone V FPGA vulnerability to SEUs and SETs has been measured and found to be approximately 100 times more vulnerable to SEUs than a similar FPGA utilizing TMR and configuration memory scrubbing and was flown in space. This is the first known instance where an Intel FPGAs vulnerability to SEUs and SETs has been measured. This is also the first known instance where a comparison has been made between the 14MeV neutron environment at Sandia National Laboratories Ion Beam Laboratory and the radiation environment experienced on a satellite for the purpose of comparing SEU rates. This work was discussed in "Intel Cyclone V FPGA Radiation Vulnerability" and submitted to the IEEE Transactions on Device and Materials Reliability Journal [39].

7.1 Contributions

A new redundancy approach to Single Event Upset (SEU) and Single Event Transient (SET) radiation effects called Adaptive-Hybrid Redundancy (AHR) was pro-

posed and demonstrated to use less energy than Triple Modular Redundancy (TMR) and run faster than Temporal Software Redundancy (TSR). AHR was comprised of TMR and TSR in such a manner that allowed it to switch between TMR and TSR. AHR was shown to be more flexible than existing redundancy methods and to equip space vehicle designers, mission planners, and operators the ability to determine how much time to spend in TMR and TSR operating modes in order to optimize performance for their particular missions.

In order to evaluate AHR when compared to the TMR and TSR approaches upon which AHR was based and an approach with no mitigation strategy, a new set of equations was developed to analyze the performance of no mitigation, TMR, TSR, and AHR approaches in terms of time to complete programs and energy used to complete programs.

In order to evaluate the unmitigated, TMR, TSR, and AHR approaches, programs had to be created which could run on all architectures, but fundamental differences between the architectures necessitated the creation of distinct, but equivalent programs. The programs for the no mitigation and TMR approaches were identical. The TSR approach utilized a program that had been modified from the no mitigation and TMR approaches such that it incorporated Error Detection by Duplicated Instructions (EDDI). The TSR program was equivalent in that the final results stored to memory were identical and TSR also computed identical intermediate results as well. The AHR approach required a program that contained the TMR and TSR programs so that it could switch between the two programs. A paper titled "Adaptive-Hybrid Redundancy for Rad-Hardening" was presented at the 2019 IEEE National Aerospace and Electronics Conference and covers the performance of AHR when compared to no mitigation, TMR, and TSR approaches in the absence of errors.

The architecture chosen as the basis for TMR, TSR, and AHR was a simplified

MIPS-like architecture called Basic MIPS and was developed specifically for this research to allow for error injection. Because it was a custom designed architecture, existing benchmarks could not be used. As a result, many sets of equivalent programs were randomly generated to evaluate the architectures against programs of varying lengths and mixtures of instructions.

To facilitate hardware testing, a "memory" was created using Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) code so that the "memory" could be implemented in the logic of a Field Programmable Gate Array (FPGA). This FPGA was made to act as memory emulator called a "memulator"; however, the memulator functioned as more than a simple memory. The aforementioned compiler which was tasked with creating the programs for the unmitigated, TMR, TSR, and AHR approaches also created the VHDL encoded memulator for each program and added error detection code for the TMR and TSR memulators. The error detection code is capable of detecting errors in the flow of program control in the TMR and TSR approaches because the compiler has perfect knowledge about how the program should execute and is able to add error checking accordingly. The purpose of the error detection was to provide an error signal that was immune to the effects of SEUs and SETs by making it external to the processor.

The memulator was also physically separated from the processor by placing it on a separate FPGA so that Hardware-in-the-Loop simulations would allow measurements of energy usage for the processor only that would not include the energy used by the memulator. Early HITL simulations co-located the processor and memulator. The results of these early HITL simulations contradicted the expectations of relative performance for TMR and TSR that were established in literature. After further analysis, it was determined that co-locating the processor and memulator was partly to blame for this contradiction and future HITL simulations will physically separate

the memulator from the processor.

7.2 Future Work

Immediate future work consists of successfully implementing Basic MIPS, TMR MIPS, TSR MIPS, and AHR MIPS on an FPGA with associated memulators on a separate FPGA by overcoming the hardware issues discussed in Sections 4.5.2 and 4.5.4. Completion of this work will enable comparisons of HITL simulations to software simulations as well as enable error injection into HITL simulations.

Another area of future work is to re-examine the boundary boxes in Figures 54 to 63 to define a more precise boundary region. This could be accomplished by analyzing every possible error injection for every single program rather than just the edge cases (TMR Type A Early/Late, TMR Type B-Best/Worst Early/Late, TSR Best/Worst).

The next step for AHR will be expanding AHR to include more redundancy methods from the literature. This will further expand the flexibility that satellite designers, mission planners, and operators will have when trading between performance, energy efficiency, and robustness to SEUs and SETs. For example, a more comprehensive TSR approach utilizing signature detection could be implemented in AHR to provide a TSR approach that uses only slightly more time and energy than EDDI, but provides a similar level of protection against program counter errors as traditional Dual Modular Redundancy (DMR) or TMR approaches. AHR could also add DMR in the lower radiation environments when it is expected that encountering an error and then rolling back the processor to a previous save restore point would be infrequent enough that DMR would save energy over TMR. When error rates are high, but insufficient to cause multiple processor errors, TMR outperforms DMR because DMR would experience frequent rollbacks which would greatly increase runtime and energy usage while TMR would quickly correct the error and move on. In low radiation

environments, TMR still consumes 1.5 times more energy than DMR (because it has three processors instead of two), but DMR is not expected to have to roll back often, and would therefore not suffer as great a time or energy penalty. Additionally, DMR would offer similar energy performance to EDDI because both involve duplication (not triplication as was the case for TMR), but run at TMR speeds (because there is still the overhead of the voter/comparator).

Another possibility for future work is to implement AHR on an ASIC in such a way that the processor and memulator are physically separated so that the memulator could be physically shielded from radiation and voltage and current measurements for the processor could be made independently from the memulator. This would greatly ease the hardware issues encountered with attempting to implement the processor and memulator on separate FPGAs.

Future work may also examine the implementation of AHR on COTS processors. Such an approach may also make use of AHR operating modes that are based on redundancy approaches designed to work in super-scalar architectures. Alternatively, an approach of this nature could seek to force COTS processors to operate in lockstep to facilitate more historical/traditional redundancy methods.

Future work should also examine radiation testing for TMR, TSR, and AHR in a laboratory setting to determine how these architectures respond to real SEUs and SETs rather than errors injected into simulations and analyses. This would be a true test of AHRs performance when compared to TMR and TSR. These tests would also make full use of the memulator's error detection capabilities.

Appendix A. AHR MIPS Architecture Detailed Diagrams

Figure 69 provides a detailed view of how the Adaptive-Hybrid Redundancy (AHR) Controller exercises control over the various signals between the Basic MIPS processors, TMR Voter, and Memulator through the use of multiplexers. Figure 70 is a more detailed version of Figure 10 in Section 3.5.2 and shows all of the signals between the Basic MIPS processors, TMR Voter, AHR Controller, and Memulator.

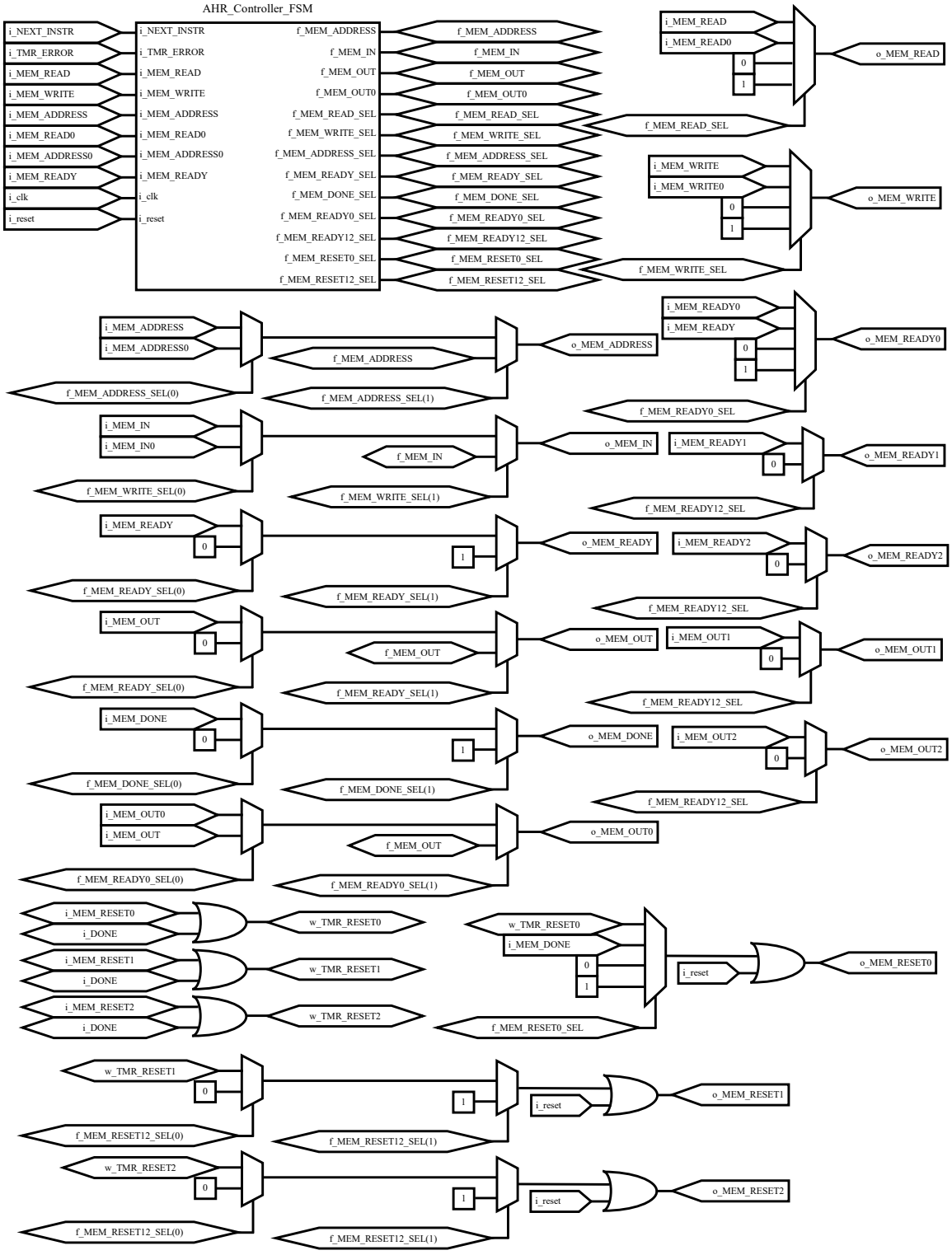

Figure 69. AHR Controller Detailed Block Diagram

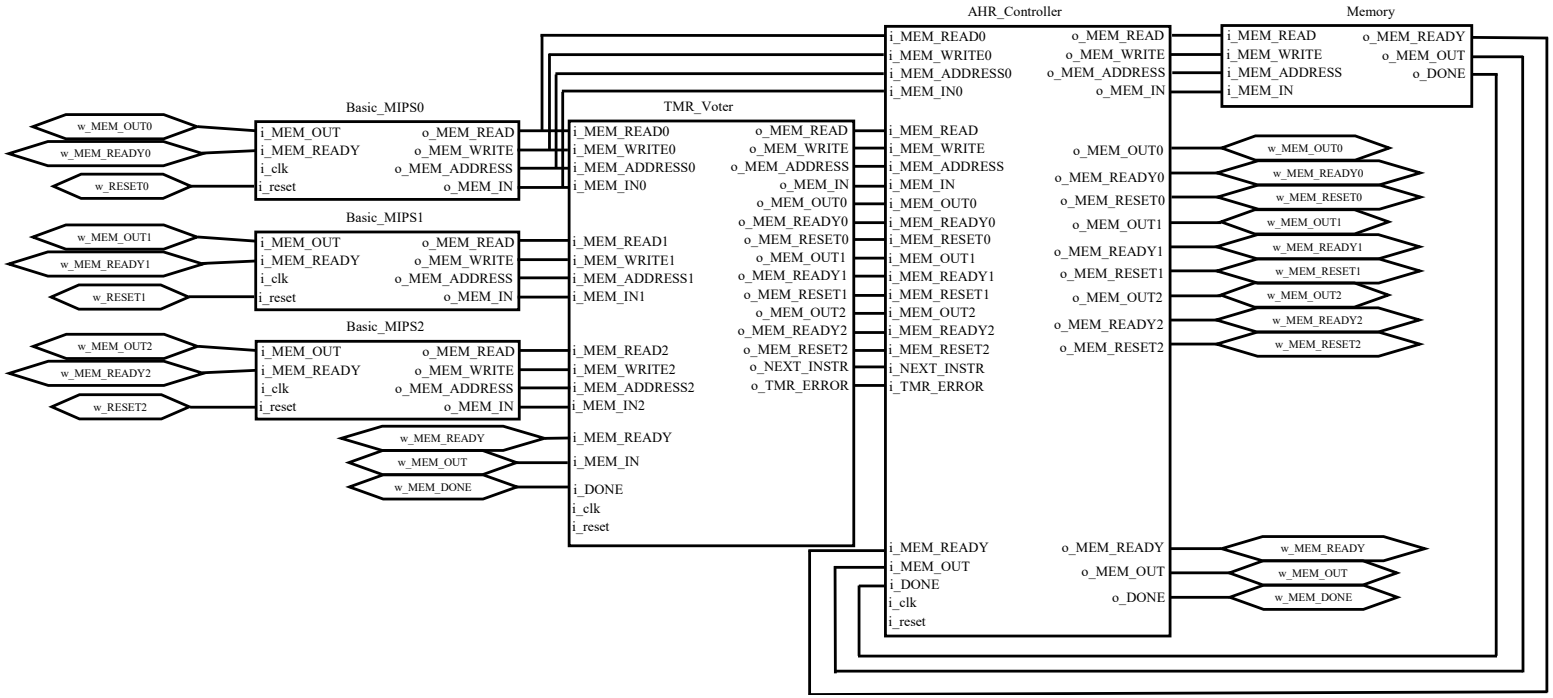

Figure 70. AHR MIPS Detailed Block Diagram

Bibliography

1. *Proceedings of the 29th International Symposium on Computer Architecture*. IEEE, May 2002.

2. *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, June 2007.

3. C. L. Axness, H. T. Weaver, J. S. Fu, R. Koga, and W. A. Kolasinski, "Mechanisms leading to single event upset," *IEEE Transactions on Nuclear Science*, vol. 33, no. 6, pp. 1577–1580, Dec 1986.

4. N. Battezzati, S. Gerardin, A. Manuzzato, D. Merodio, A. Paccagnella, C. Poivey, L. Sterpone, and M. Violante, "Methodologies to study frequency-dependent single event effects sensitivity in flash-based fpgas," *IEEE Transactions on Nuclear Science*, vol. 56, no. 6, pp. 3534–3541, Dec 2009.

5. M. Berg, H. Kim, M. Friendlich, C. Perez, C. Seidleck, K. LaBel, and R. Ladbury, "Seu analysis of complex circuits implemented in actel rtax-s fpga devices," *IEEE Transactions on Nuclear Science*, vol. 58, no. 3, pp. 1015–1022, Jun 2011.

6. R. E. Bickel, "Fault tolerant processing architecture," Patent US 2003/0 061 535 A1, Mar 27, 2003.

7. ——, "Fault tolerant processing architecture," Patent US 6,938,183 B2, Aug 30, 2005.

8. D. Binder, E. C. Smith, and A. B. Holman, "Satellite anomalies from galactic cosmic rays," *IEEE Transactions on Nuclear Science*, vol. 22, no. 6, pp. 2675–2680, Dec 1975.

9. M. A. Breuer and A. J. Carlan, "State-of-the art assessment of testing and testability of custom lsi-vlsi circuits. volume iv. test generation," Oct 1982.

10. ——, "State-of-the-art assessment of testing and testability of custom lsi-vlsi circuits. volume vi. redundancy, testing circuits, and codes," Oct 1982.

11. T. Calin, M. Nicolaidis, and R. Velazco, "Upset hardened memory design for submicron cmos technology," *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, pp. 2874–2878, Dec 1996.

12. A. Corporation, *Design Techniques for Radiation-Hardened FPGAs*, 1 Enterprise, Aliso Viejo, CA 92656 USA, 9 1997.

13. D. R. Czajkowski, "Seu and sefi fault tolerant computer," Patent 7,260,742, Aug, 2007.

14. ——, "Fault tolerant computer," Patent 7,318,169, Jan, 2008.

15. S. E. Diehl-Nagle, J. E. Vinson, and E. L. Peterson, "Single event upset rate predictions for complex logic systems," *IEEE Transactions on Nuclear Science*, vol. 31, no. 6, pp. 1132–1138, Dec 1984.

16. P. E. Dodd and F. W. Sexton, "Critical charge concepts for cmos srams," *IEEE Transactions on Nuclear Science*, vol. 42, no. 6, pp. 1764–1771, Dec 1995.

17. P. E. Dodd and L. W. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 583–602, June 2003.

18. E. Electronics, *EA1040 Series Datasheet*, 11F-2. No.150 Jian Yi Road, New Taipei City, Taiwan, 2016.

19. M. D. Ercegovac, A. Avižienis, and T. Lang, "Specification and design methodologies for high-speed fault-tolerant array algorithms and structures for vlsi," Jun 1987.

20. D. C. Espinosa, A. Geist, D. J. Petrick, T. P. Flatley, J. C. Hosler, G. A. Crum, and M. Buenfil, "Radiation-hardened processing system," Patent 2011/0 107 158 A1, 2011.

21. ——, "Radiation-hardened processing system," Patent 8,484,509 B2, 2013.

22. T. P. Flatley, "Radiation-hardened hybrid processor," Patent 2011/0 078 498 A1, 2011.

23. C. Frenkel, J.-D. Legat, and D. Bol, "A partial reconfiguration-based scheme to mitigate multiple-bit upsets for fpgas in low-cost space applications," in *2015 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE, Jun 2015, pp. 1–7.

24. A. L. Friedman, B. Lawton, K. R. Hotelling, J. C. Pickel, V. H. Strahan, and K. Loree, "Single event upset in combinatorial and sequential current mode logic," *IEEE Transactions on Nuclear Science*, vol. 32, no. 6, pp. 4216–4218, Dec 1985.

25. E. Fuller, P. Blain, M. Caffrey, C. Carmichael, N. Khalsa, and A. Salazar, "Radiation test results of the virtex fpga and zbt sram for space based reconfigurable computing," pp. 1–8, 1999.

26. K. F. Galloway and R. D. Schrimpf, *Interaction of Radiation with Semiconductor Devices*, Nov 2012, pp. 79–91.

27. A. Geist, T. P. Flatley, M. R. Lin, and D. J. Petrick, "Radiation-hardened hybrid processor," Patent 2011/0 099 421 A1, 2011.

28. T. B. Getz, "Radiation induced fault detection, diagnosis, and characterization on fpgas," Master's thesis, Air Force Institute of Technology, Mar 2011.

29. K. W. Golke, H. H. L. Liu, M. S. Liu, and D. K. Nelson, "Radiation-hardened memory element with multiple delay elements," Patent US2007/0 242 537 A1, 2007.

30. M. A. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," *IEEE Micro*, vol. 23, no. 6, pp. 76–83, Nov 2003.

31. M. Grecki, "Seus tolerance in fpgas based digital llrf system for xfel," in *2012 18th IEEE-NPSS Real Time Conference*. IEEE, June 2012, pp. 1–3.

32. M. Gruss. (2014, Apr) U.S. Air Force Orders Two More GPS 3 Satellites. [Online]. Available: http://spacenews.com/40068us-air-force-orders-two-more-gps-3-satellites/

33. C. S. Guenzer, E. A. Wolicki, and R. G. Allas, "Single event upset of dynamic rams by neutrons and protons," *IEEE Transactions on Nuclear Science*, vol. 26, no. 6, pp. 5048–5052, Dec 1979.

34. T. L. Gunckel and J. S. Irvine, "Radiation hardened register file," Patent 4,199,810, 1980.

35. N. S. Hamilton, "Adaptive-Hybrid Redundancy MIPS Architecture Version 2.2," Jul 2019.

36. ——, "Basic MIPS Architecture Version 1.4," Jul 2019.

37. ——, "DE10 Pins And Connections for Basic MIPS," Jul 2019.

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

39. N. S. Hamilton, S. Graham, T. J. Carbino, and J. Petrosky, "Intel cyclone v fpga radiation vulnerability," *IEEE Transactions on Device and Materials Reliability*, 2019.

40. C. R. P. Hartman, P. K. Lala, A. M. Ali, G. S. Visweswaran, and S. Ganguly, "Fault tolerant vlsi (very large-scale integration) design using error correcting codes," Feb 1989.

41. K. J. Hass, R. K. Treece, and A. E. Giddings, "A radiation-hardened 16/32-bit microprocessor," *IEEE Transactions on Nuclear Science*, vol. 36, no. 6, pp. 2252–2257, Dec 1989.

42. R. A. Hillman and M. S. Conrad, "Self-correcting computer," Patent 7,890,799, 2011.

43. G. W. Hughes, G. J. Brucker, and R. K. Smeltzer, "Radiation-hardened n(+) gate cmos/sos," May 1981.

44. Intel, *Cyclone V Device Overview*, 2200 Mission College Blvd., Santa Clara, CA 95054-1549, May 2018.

45. X. Iturbe, B. Venu, E. Özer, and S. Das, "A triple core lock-step (tcls) arm cortex-r5 processor for safety-critical and ultra-reliable applications," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, July 2016, pp. 246–249.

46. C. M. Jeffery and R. J. Figueiredo, "A flexible approach to improving system reliability with virtual lockstep," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 1, pp. 2–15, Jan 2012.

47. K. Katsarou and Y. Tsiatouhas, "Soft error interception latch: Double node charge sharing seu tolerant design," *Electronic Letters*, vol. 51, no. 4, pp. 330–332, Feb 2015.

48. B. Kolla, P. K. Lala, and K. C. Yarlagadda, "A concurrent checking scheme for single and multibit errors in logic circuits," in *Digest of Papers. 1992 IEEE VLSI Test Symposium*. IEEE, April 1992, pp. 160–164.

49. J. Kontoleon, "Soft error recovery in simplex and triplex memory systems," *Microelectronics Reliability*, vol. 49, no. 4, pp. 410–423, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0026271408004319

50. P. l. Montesinos, W. Liu, and J. Torrellas, "Using register lifetime predictions to protect register files against soft-errors," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, June 2007, pp. 286–296.

51. K. A. Label and L. M. Cohn, "Radiation testing, characterization and qualification challenges for modern microelectronics and photonics devices and technologies," in *2008 Government Microcircuit Applications and Critical Technology Conference*. U.S. Department of Defense, Mar 2008.

52. P. K. Lala, F. Busaba, and K. C. Yarlagadda, "An approach for designing self-checking logic using residue codes," in *VLSI Test Symposium 1991. 'Chip-to-System Test Concerns for the 90's', Digest of Papers*. IEEE, April 1991, pp. 166–171.

53. P. K. Lala and H. L. Martin, "Application of error correcting codes in fault-tolerant logic design for vlsi circuits," May 1990.

54. ——, "Application of error correcting codes in fault-tolerant logic design for vlsi circuits," Aug 1992.

55. B. J. LaMares and C. Gauer, "A power-efficient design approach to radiation hardened digital circuitry using dynamically selectable triple modulo redundancy," in *2008 Military & Aerospace Programmable Logic Devices (MAPLD) Conference*, Sep. 2008.

56. D. C. Liddell and E. J. Williams, "Method and apparatus for reducing the effects of hardware faults in a computer system employing multiple central processing modules," Patent 5,627,965, 1997.

57. F. Lima, C. Carmichaell, J. Fabula, R. Padovanil, and R. Reis, "A fault injection analysis of virtex fpga tmr design methodology," in *2001 6th European Conference on Radiation and Its Effects on Components and Systems*. IEEE, Sep 2001, pp. 275–282.

58. A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors - a survey," *IEEE Transactions on Computers*, vol. 37, no. 2, pp. 160–174, Feb 1988.

59. D. G. Mahmoud, G. I. Alkady, H. H. Amer, R. M. Daoud, I. Adly, Y. Essam, H. A. Ismail, and K. N. Sorour, "Fault secure fpga-based tmr voter," in *2018 7th Mediterranean Conference on Embedded Computing*. IEEE, June 2018, pp. 1–4.

60. L. W. Massengill, A. E. Baranski, D. O. V. Nort, J. Meng, and B. L. Bhuva, "Analysis of single-event effects in combinational logic-simulation of the am2901 bitslice processor," *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2609–2615, Dec 2000.

61. P. J. McNulty, G. E. Farrell, and R. C. Wyatt, "Upset phenomena induced by energetic protons and electrons," *IEEE Transactions on Nuclear Science*, vol. 27, no. 6, pp. 1516–1522, Dec 1980.

62. S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluatation of redundant multithreading alternatives," in *Proceedings of the 29th International Symposium on Computer Architecture.* IEEE, May 2002, pp. 99–110.

63. N. Nakka, K. Pattabiraman, and R. K. Iyer, "Processor-level selective replication," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* IEEE, June 2007, pp. 286–296.

64. D. K. Nelson, K. W. Golke, H. H. L. Liu, and M. S. Liu, "Radiation-hardened memory element with multiple delay elements," Patent 8,767,444, 2014.

65. T. S. Nidhin, A. Battacharyya, R. P. Behera, T. Jayanthi, and K. Velusamy, "Understanding radiation effects in sram-based field programmable gate arrays for implementing instrumentation and control systems of nuclear power plants," *Nuclear Engineering and Technology*, vol. 49, no. 8, pp. 1589–1599, Dec 2017.

66. E. Normand, "Single-event effects in avionics," *IEEE Transactions on Nuclear Science*, vol. 43, no. 2, pp. 461–474, Apr 1996.

67. E. Normand, D. L. Oberg, J. L. Wert, J. D. Ness, P. P. Majewski, S. Wender, and A. Gavron, "Single event upset and charge collection measurements using high energy protons and neutrons," *IEEE Transactions on Nuclear Science*, vol. 41, no. 6, pp. 2203–2209, Dec 1994.

68. G. Northrop, R. Averill, K. Barkley, S. Carey, Y. Chan, Y. H. Chan, M. Check, D. Hoffman, W. Huott, B. Krumm, C. Krygowski, J. Liptay, M. Mayo, T. McNamara, T. McPherson, E. Schwarz, L. S. T. Siegel, C. Webb, D. Webber, and P. Williams, "609 mhz g5 s/399 microprocessor," in *1999 IEEE International Solid-State Circuits Conference. Digest of Technical Papers. ISSCC. First Edition (Cat. No.99CH36278)*, Feb 1999, pp. 88–89.

69. N. Oh and E. J. McCluskey, "Low energy error detection technique using procedure call duplication," 01 2001.

70. N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, no. 2, pp. 111 – 122, Mar 2002.

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

72. J. Ohlsson and M. Rimén, "Implicit signature checking," in *25th International Symposium on Fault-Tolerant Computing, Digest of Papers*. IEEE, Jun 1995, pp. 218–227.

73. 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," *IEEE Transactions on Nuclear Science*, vol. 56, no. 6, pp. 3519–3526, Dec 2009.

74. E. Özer, M. M. Ghahroodi, and D. Bull, "Seu and set-tolerant arm cortex-r4 cpu for space and avionics applications," in *2nd Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale - MEDIAN 2013*, May 2013.

75. D. K. Pradhan, "Fault-tolerant architectures for multiprocessor and vlsi-based systems," Sep 1992.

76. I. K. Proudler, "Algorithmic fault tolerance," Sep 1988.

77. J. Ray, J. C. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," in *Proceedings of the 34th Annual ACM/IEEEE International Symposium on Microarchitecture*. IEEE, Dec 2001, pp. 214–224.

78. S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of the 27th International Symposium on Computer Architecture*. IEEE, June 2000, pp. 25–36.

79. G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *International Symposium on Code Generation and Optimization, 2005*. IEEE, Mar. 2005.

80. G. A. Reis, J. Chang, and D. I. August, "Automatic instruction-level software-only recovery," *IEEE Micro*, vol. 27, no. 1, pp. 36 – 47, May 2007.

81. S. Rezgui, J. J. Wang, Y. Sun, B. Cronquist, and J. McCollum, "Configuration and routing effects on the set propagation in flash-based fpgas," *IEEE Transactions on Nuclear Science*, vol. 55, no. 6, pp. 3328–3335, Dec 2008.

82. S. Rezgui, R. Won, and J. Tien, "Set characterization and mitigation in 65-nm cmos test structures," *IEEE Transactions on Nuclear Science*, vol. 59, no. 4, pp. 851–859, June 2012.

83. H. S. Riccardo Mariani, Thomas Kuschel, "A flexible microcontroller architecture for fail-safe and fail-operational systems," in *Proceedings of the HiPEAC Workshop on Design for Reliability*. HiPEAC, January 2010.

84. E. Rotenberg, "Ar-smt: A microarchitectural approach to fault tolerance in microprocessors," in *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*. IEEE, Jun 1999, pp. 84–91.

85. J.-P. Schoellkopf, "Simple technique shields embedded sram from soft errors," *Electronic Component News*, vol. 48, no. 2, pp. 53–54, Feb 2004.

86. 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, Mar 2000.

87. A. D. Singh and F. G. Gray, "Periodically self restoring redundant systems for vlsi based highly reliable design," Jul 1986.

88. L. Sterpone, B. Du, and S. Azimi, "Radiation-induced single event transients modeling and testing on nanometric flash-based technologies," in *Proceedings of the 26th European Symposium on Reliability of Electron Devices, Failure, Failure Physics, and Analysis*, Oct. 2015.

89. L. Sterpone, N. Battezzati, and V. Ferlet-Cavrois, "Analysis of set propagation in flash-based fpgas by means of electrical pulse injection," *IEEE Transactions on Nuclear Science*, vol. 57, no. 4, pp. 1820–1826, Aug 2010.

90. L. Sterpone, N. Battezzati, F. L. Kastensmidt, and R. Chipana, "An analytical model of the propagation induced pulse broadening (pipb) effects on single event transient in flash-based fpgas," *IEEE Transactions on Nuclear Science*, vol. 58, no. 5, pp. 2333–2340, Oct 2011.

91. L. Sterpone and B. Du, "Analysis and mitigation of single event effects on flash-based fpgas," in *2014 19th IEEE European Test Symposium (ETS)*. IEEE, May 2014, pp. 1–6.

92. M. Straka, J. Kastil, and Z. Kotasek, "Fault tolerant structure for sram-based fpga via partial dynamic reconfiguration," in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. IEEE, Sep 2010, pp. 365–372.

93. J. Suh, M. Manoocherhri, M. Annavaram, and M. Dubois, "Soft error benchmarking of l2 caches with parma," in *Proceedings of the ACM SIGMETRICS*

Joint INternational Conference on Measurement and Modeling of Computer Systems. ACM, June 2011, pp. 85–96.

94. J. Tabero, A. Regadío, C. Pérez, Jesús, Pazos, P. Reviriego, A. Sánchez-Macian, and J. A. Maestro, "Modular fault tolerant processor architecture on a soc for space," *Microelectronic Reliability*, vol. 83, pp. 84–90, Apr 2018.

95. Y. Tamir, "Fault tolerance for vlsi multicomputers," Ph.D. dissertation, University of California, Berkeley, Aug 1985.

96. K. Technologies, *N2870A-Series and N2894A Passive Probes User's Guide*, 1400 Fountaingrove Parkway, Santa Rosa, CA 95403, Apr 2016.

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

98. ——, *Keysight N2870/1A High Sensitivity Current Probes User's Guide*, 1400 Fountaingrove Parkway, Santa Rosa, CA 95403, Apr 2017.

99. Terasic, *SoCKit_Rev_B_Schematic*, 9F. No.176 Sec.2 Gongdao 5th Rd, Hsinchu City, Taiwan, Apr 2013.

100. ——, *THDB-HTG Revision B Schematic*, 9F. No.176 Sec.2 Gongdao 5th Rd, Hsinchu City, Taiwan, Apr 2014.

101. M. P. Tokponnon, M. Lobelle, and E. C. Ezin, "Entirely protecting operating systems against transient errors in space environment," Aug 2017.

102. T. Uemura, Y. Tosaka, H. Matsuyama, K. Shono, C. J. Uchibori, K. Takahisa, M. Fukuda, and K. Hatanaka, "Seila: Soft error immune latch for mitigating multi-node-seu and local-clock-set," in *2010 IEEE International Reliability Physics Symposium*. IEEE, May 2010, pp. 218–223.

103. R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *Proceedings of the 9th IEEE On-Line Testing Symposium*. IEEE, Jul 2003, pp. 137–143.

104. T. N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient fault recovery using simultaneous multithreading," in *Proceedings of the 29th International Symposium on Computer Architecture*. IEEE, May 2002, pp. 99–110.

105. J. T. Wallmark and S. M. Marcus, "Minimum size and maximum packing density of nonredundant semiconductor devices," *Proceedings of the IRE*, vol. 50, no. 3, pp. 286–298, March 1962.

106. S. Wang, J. Hu, and S. G. Ziavras, "Self-adaptive data caches for soft-error reliability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 8, pp. 1503–1507, Aug 2008.

107. N. H. E. Weste and D. M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective Fourth Edition*. Addison Wesley, 2011.

108. M. Wirthlin, "High-reliability fpga-based systems: Space, high-energy physics, and beyond," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 379–389, Mar 2015.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
12–09–2019	Doctoral Dissertation	Sept 2016 — Sept 2019

4. TITLE AND SUBTITLE

Adaptive-Hybrid Redundancy for Radiation Hardening

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Hamilton, Nicolas S, Maj, USAF

5d. PROJECT NUMBER

18G169C

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-DS-19-S-005

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

An Adaptive-Hybrid Redundancy (AHR) mitigation strategy is proposed to mitigate the effects of Single Event Upset (SEU) and Single Event Transient (SET) radiation effects. AHR is adaptive because it switches between Triple Modular Redundancy (TMR) and Temporal Software Redundancy (TSR). AHR is hybrid because it uses hardware and software redundancy. AHR is demonstrated to run faster than TSR and use less energy than TMR. Furthermore, AHR allows space vehicle designers, mission planners, and operators the flexibility to determine how much time is spent in TMR and TSR. TMR mode provides faster processing at the expense of greater energy usage. TSR mode uses less energy at the expense of processing speed. AHR allows the user to determine the optimal balance between these modes based on their mission needs and changes can be made even after the space vehicle is operational. Radiation testing was performed to determine the SEU injection rate for simulations and analyses. A Field Programmable Gate Array (FPGA) was used to expedite testing in hardware.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Scott Graham, AFIT/ENG
U	U	U	UU	265	19b. TELEPHONE NUMBER *(include area code)* (937) 255-6565 x4581; scott.graham@afit.edu