



**TRIPLE MODULAR REDUNDANCY
MIPS ARCHITECTURE
VERSION 1.4**

TECHNICAL REPORT

Nicolas Hamilton, Major, USAF
AFIT/EN/TR-19-03

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Disclaimer: The views expressed in this paper are those of the authors, and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U. S. Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States. This document has been approved for public release; distribution unlimited, case number 88ABW-2019-5331

Abstract

This report describes in detail the architecture of a Triple Modular Redundancy (TMR) MIPS processor based upon the Basic MIPS processor [1]. The TMR MIPS processor is used for Adaptive-Hybrid Redundancy research. There may be many other applications for the TMR MIPS processor beyond this specific research area.

Table of Contents

	Page
Abstract	ii
List of Figures	iv
List of Tables	v
I. Introduction	1
II. TMR Voter	6
2.1 Normal TMR Voter Operation	8
2.2 ERR0 TMR Voter Operation	16
2.3 ERR1 TMR Voter Operation	24
2.4 SAVE TMR Voter Operation	30
Appendix A. TMR Voter VHDL Code	37
Appendix B. Version History	104
Bibliography	106

List of Figures

Figure		Page
1	TMR MIPS High-Level Block Diagram	2
2	TMR MIPS Detailed Block Diagram	5

List of Tables

Table		Page
1	TMR Voter Inputs and Outputs	7
2	Voter Normal Operation States - Most Instructions.....	9
3	Normal Voter Outputs	14
4	ERR0 Voter Operation	17
5	ERR0 Voter Outputs.....	21
6	ERR1 Voter Operation	25
7	ERR1 Voter Outputs.....	28
8	SAVE Voter Operation	31
9	SAVE Voter Outputs.....	35

I. Introduction

The goal of this project was to utilize the Basic MIPS architecture in a Triple Modular Redundancy (TMR) configuration. In TMR, three processors utilize the same inputs and produce identical outputs under error free circumstances. Figure 1 shows a simplified schematic of a TMR architecture. If one processor disagrees with the other two, the disagreeing processor is reset and its internal state set to match the states of the two agreeing processors. If a multiple processor error occurs, all three processors are reset and restored to a previously saved system state. The internal state of the processors are periodically stored as a precaution against multiple processor errors. This saved internal state is referred to as a save/restore point. At any given time, TMR must maintain two save/restore points. One is always more recent than the other and considered active. The active save/restore point is used to restore the processors if a multiple processor error occurs. When a new save/restore point is created, it is saved to the least recent or inactive save/restore point. If the save/restore point is successfully created, it becomes the active save/restore point.

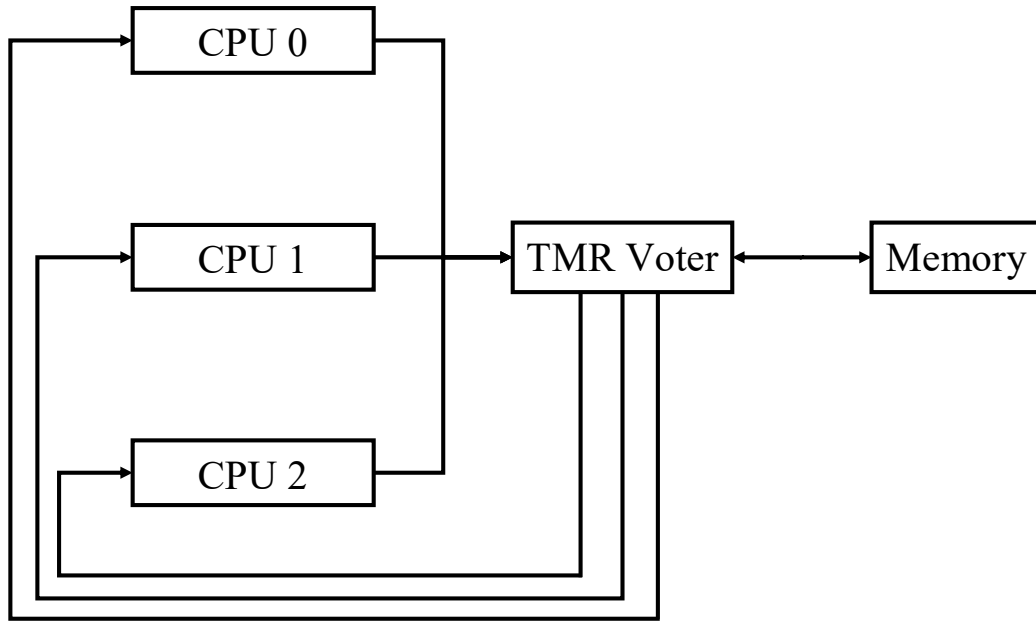


Figure 1. TMR MIPS High-Level Block Diagram

A TMR Voter was created to take inputs from and provide outputs to three Basic MIPS processors. The TMR Voter is an elaborate Finite State Machine (FSM) that continually examines each Basic MIPS processor's memory requests. Assuming all three processors agree on the type of operation to be performed (read or write), the address to be accessed for both read and write operations, and the data to be sent for write operations only, the TMR Voter then performs the requested memory operation and returns the requested data for read operations and also the memory "Ready" signal when read and write operations are successfully completed. If one of the three processors disagrees with the other two on the type of operation to be performed, the address, and/or the data to be written (for a write operation), the TMR Voter enters a special error state called ERR0.

In this ERR0 state, the processor that disagreed is reset, then MIPS commands

are sequentially issued to all three processors so that the contents of all registers in the two processors that agree are written to the registers in the processor that disagrees one at a time. If the two processors that seemingly agree when ERR0 is entered ever disagree on any of the register values or the next address to be accessed, another special error state called ERR1 is entered.

ERR1 can also be entered if all three processors disagree on the address (for read or write operations) or data to be written (for write operations only) during normal voter operation. In ERR1, the voter resets all three processors, locates the active save/restore point in memory, then issues sequential commands to all three processors to load all register values from the save/restore point in memory into the registers. The existence of a save/restore point means there is also another special FSM state called SAVE during which the save/restore point is created. Two save/restore points are maintained and only one of those two are designated as “active.” The active one is the most recent one. The inactive one is the least recent one.

The SAVE state is entered after a predefined number of instructions are read from memory by the three processors during normal voter operation. In the SAVE state, the voter locates the active save/restore point in memory and then determines the inactive save/restore point. The voter then issues a series of store commands to all three processors in order to store each register value to the inactive save/restore point in memory. So long as the processors stay in sync, are accessing the same addresses, and at least two of the three processors agree on the value of each register, the value of each register is stored to memory. Once all registers have been stored to memory, the voter updates the active save/restore point to point to the save/restore point that was just created. If any of the processors drops out of sync or all three processors disagree on the value of a register during this process, the voter enters the ERR1 state.

The memory for the save/restore point is constructed in such a way that save/restore point 0 is comprised of memory locations 0 through 31 and save/restore point 1 is comprised of memory locations 32 through 63. Memory location 64 is used to determine which of the two save/restore points is active. The least significant bit being 0 means 0 is active and the least significant bit being 1 signifies 1 is active. Memory locations 0 through 30 and 32 through 62 store the 31 user defined register values at the time the save/restore point is created. Memory locations 31 and 63 store the program counter value when the save/restore point is created. One should note however that these memory locations are really offset to be at the end of the MIPS program. If the program has instructions and temporary and permanent memory storage that occupy memory locations 0 through 99, then the save/restore point memory occupies 100 through 164. Because MIPS indices point to bytes and not words, each location previously mentioned must be multiplied by 4 when referencing them in MIPS Instructions (i.e. 0 = 0, 1 = 4, 2 = 8, and so on).

Figure 2 shows a more detailed view of TMR MIPS and how the three processors connect to the TMR Voter and how the TMR Voter connects to memory. Note that the w_DONE signal supplied by memory to the TMR Voter is used to reset the TMR Voter and the three processors upon completion of a program stored in memory.

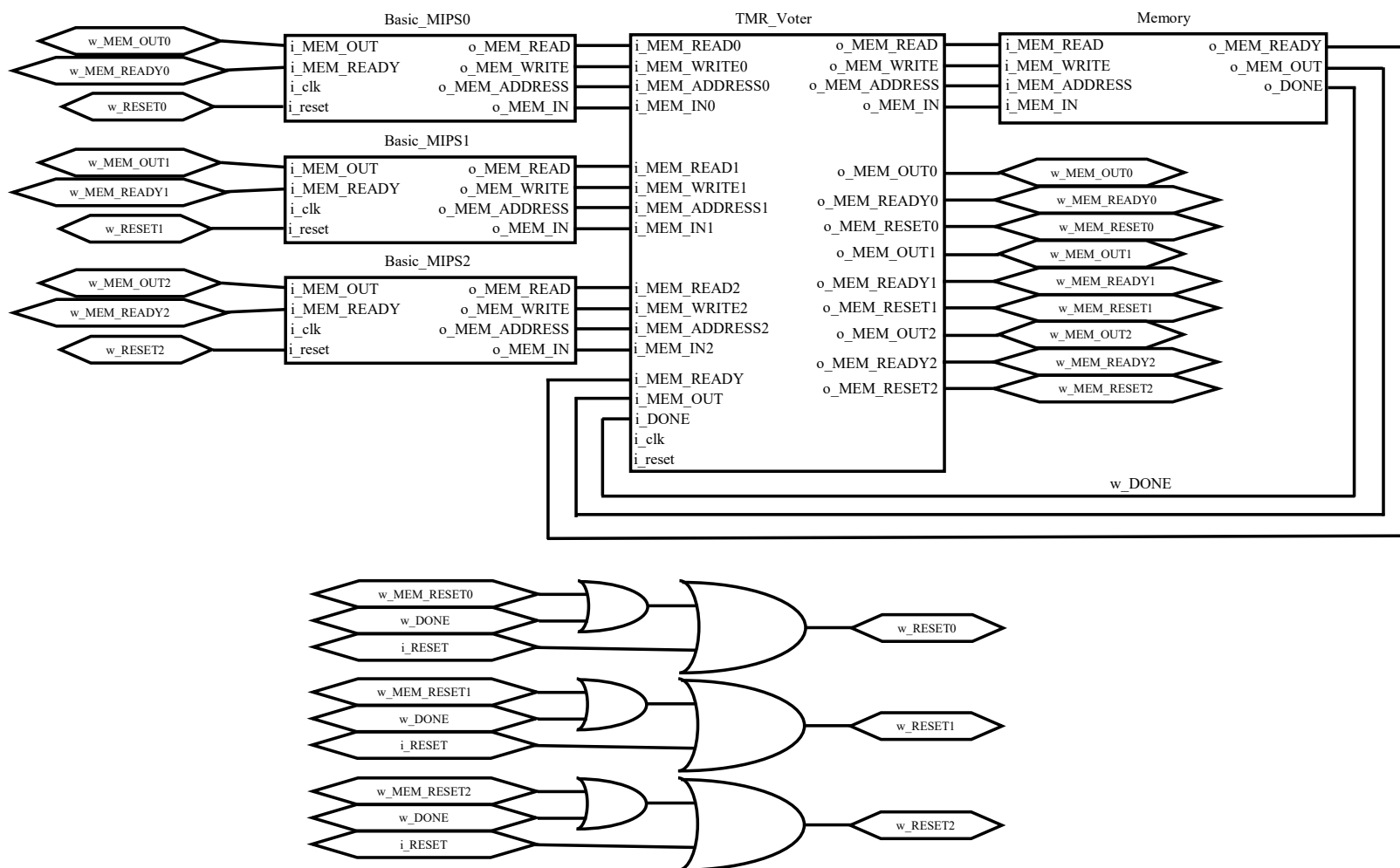


Figure 2. TMR MIPS Detailed Block Diagram

II. TMR Voter

TMR Voter inputs and outputs are shown in Table 1.

Table 1. TMR Voter Inputs and Outputs

Name	Type	Width	Description
MEM_READ	Out	1	1 when the voter is attempting to read from memory
MEM_WRITE	Out	1	1 when the voter is attempting to write to memory
MEM_ADDRESS	Out	32	Address from which the voter is attempting to read or to which the voter is attempting to write
MEM_DATA_IN	Out	32	Data the voter is attempting to write to memory
MEM_DATA_OUT	In	32	Data memory provides the voter in response to a read request from the voter
MEM_READY	In	1	1 when the data the voter is attempting to read is stable or when the data the voter attempted to store has been successfully written to memory
READ0 READ1 READ2	In	1	1 when a processor is attempting to read data from the voter
WRITE0 WRITE1 WRITE2	In	1	1 when a processor is attempting to write data to the voter
ADDRESS0 ADDRESS1 ADDRESS2	In	32	Address from which a processor is attempting to read or to which a processor is attempting to write
DATA0 DATA1 DATA2	In	32	Data a processor is attempting to write to the voter
MEM_OUT0 MEM_OUT1 MEM_OUT2	Out	32	Data or instruction the voter is providing to the processors in response to a read request from the processors
MEM_READY0 MEM_READY1 MEM_READY2	Out	1	1 when the voter is ready to provide the data or instruction requested by a processor
RESET0 RESET1 RESET2	Out	32	1 when the voter is resetting a processor

2.1 Normal TMR Voter Operation

During normal operation, the voter proceeds through states FSM_0 through FSM_12. The states and their operation are illustrated in Table 2. If the operation is a load word instruction, normal operation proceeds through states FSM_0 to FSM_7a before returning to FSM_0. If the operation is a store word instruction, normal operation proceeds through states FSM_0 to FSM_3a and FSM_8 to FSM_12 before returning to FSM_0. If the operation is neither a load or store word instruction, normal operation proceeds through states FSM_0 to FSM_3a before returning to FSM_0. If an error state is entered, the voter returns from the error state or transitions from ERR0 to ERR1 as shown in toward the end of Table 2. The voter also determines when to enter the save/restore point creation state FSM_SAVE in order to create a save/restore point. This state returns to the FSM_0 state or progresses to the FSM_ERR1 state as shown at the bottom of Table 2.

Table 3 shows the outputs of the finite state machine when operating normally. If any of the voter output signals from Table 1 do not appear in the outputs for a particular state, then the output remains unchanged from the previous state in which the output was changed.

Table 2. Voter Normal Operation States - Most Instructions

Current State	Next State	Transition Condition	Description
FSM_0	FSM_1	(READ0 = 1 and READ1 = 1) or (READ1 = 1 and READ2 = 1) or (READ0 = 1 or READ2 = 1)	Idle state Waiting for at least two of three processors to try to read from memory
FSM_0	FSM_ERR1	TIMEOUT	One or more processors did not attempt to read from memory in a timely fashion
FSM_1	FSM_2	READ0 = 1 and READ1 = 1 and READ2 = 1 and ADDRESS0 = ADDRESS1 and ADDRESS0 = ADDRESS2	Determine if processors are all attempting to read the same instruction from memory
FSM_1	FSM_ERR0	1. READ0 = 1 and READ1 = 1 and ADDRESS0 = ADDRESS1 2. READ1 = 1 and READ2 = 1 and ADDRESS1 = ADDRESS2 3. READ0 = 1 and READ2 = 1 and ADDRESS0 = ADDRESS2	Two of three processors agree
FSM_1	FSM_ERR1	ERR1	All three processors disagree
FSM_1	FSM_SAVE	SAVE_POINT	All three processors agree and a save point has been reached
FSM_2	FSM_3	MEM_READY = 1	Read data from memory Wait for memory to be ready
FSM_3	FSM_3a	READ0 = 0 and READ1 = 0 and READ2 = 0	Return instruction from memory to processors
FSM_3	FSM_ERR1	TIMEOUT	One or more processors did

Table 2 – Continued on next page

Table 2 – *Continued from previous page*

Current State	Next State	Transition Condition	Description
			not read the instruction in a timely fashion
FSM_3a	FSM_0	MEM_READY = 0	Wait for memory ready signal to return to 0. Instruction was not a load or store word
FSM_3a	FSM_4	MEM_READY = 0	Wait for memory ready signal to return to 0. Instruction was a load word
FSM_3a	FSM_8	MEM_READY = 0	Wait for memory ready signal to return to 0. Instruction was a store word
FSM_4	FSM_5	(READ0 = 1 and READ1 = 1) or (READ1 = 1 and READ2 = 1) or (READ0 = 1 or READ2 = 1)	Load word operation. Waiting for at least two of three processors to try to read from memory
FSM_4	FSM_ERR1	TIMEOUT	One or more processors did not read from memory in a timely fashion
FSM_5	FSM_6	READ0 = 1 and READ1 = 1 and READ2 = 1 and ADDRESS0 = ADDRESS1 and ADDRESS0 = ADDRESS2	Determine if processors are all attempting to read the same data from memory
FSM_5	FSM_5a	1. READ0 = 1 and READ1 = 1 and ADDRESS0 = ADDRESS1 2. READ1 = 1 and READ2 = 1 and ADDRESS1 = ADDRESS2 3. READ0 = 1 and READ2 = 1 and	Two of three processors agree

Table 2 – *Continued on next page*

Table 2 – *Continued from previous page*

Current State	Next State	Transition Condition	Description
		ADDRESS0 = ADDRESS2	
FSM_5	FSM_ERR1	ERR1	All three processors disagree
FSM_5a	FSM_ERR0	1. READ0 = 0 and READ1 = 0 and 2. READ1 = 0 and READ2 = 0 and 3. READ0 = 0 and READ2 = 0 and	Wait for the read signals of the two processors in agreement to return to 0
FSM_5a	FSM_ERR1	TIMEOUT	The processors in agreement did not return their read signals to 0 in a timely fashion
FSM_6	FSM_7	MEM_READY = 1	Read data from memory Wait for memory to be ready
FSM_7	FSM_7a	READ0 = 0 and READ1 = 0 and READ2 = 0	Return data from memory to processors and wait for read signals to return to 0
FSM_7	FSM_ERR1	TIMEOUT	The processors in agreement did not return their read signals to 0 in a timely fashion
FSM_7a	FSM_0	MEM_READY = 0	Wait for memory ready signal to return to 0. End of load word FSM states
FSM_8	FSM_9	(WRITE0 = 1 and WRITE1 = 1) or (WRITE1 = 1 and WRITE2 = 1) or (WRITE0 = 1 or WRITE2 = 1)	Waiting for at least two of three processors to try to write to memory
FSM_8	FSM_ERR1	TIMEOUT	One or more processors did not attempt to write to memory in a timely fashion
FSM_9	FSM_10	WRITE0 = 1 and WRITE1 = 1 and WRITE2 = 1 and	Determine if processors are all attempting to write the same

Table 2 – *Continued on next page*

Table 2 – *Continued from previous page*

Current State	Next State	Transition Condition	Description
		ADDRESS0 = ADDRESS1 and ADDRESS0 = ADDRESS2 and DATA0 = DATA1 and DATA1 = DATA2	data to the same address
FSM_9	FSM_9a	WRITE0 = 1 and WRITE1 = 1 and WRITE2 = 1 and ADDRESS0 = ADDRESS1 and ADDRESS0 = ADDRESS2 and (DATA0 = DATA1 or DATA1 = DATA2 or DATA0 = DATA2)	If all processors are attempting to write to the same address, determine if 2 of 3 are writing the same data
FSM_9	FSM_9a	1. WRITE0 = 1 and WRITE1 = 1 and ADDRESS0 = ADDRESS1 and DATA0 = DATA1 2. WRITE1 = 1 and WRITE2 = 1 and ADDRESS1 = ADDRESS2 and DATA1 = DATA2 3. WRITE0 = 1 and WRITE2 = 1 and ADDRESS0 = ADDRESS2 and DATA0 = DATA2	If two of three processors are attempting to write to the same data to the same address
FSM_9	FSM_ERR1	ERR1	All three processors disagree
FSM_9a	FSM_ERR0	1. WRITE0 = 0 and WRITE1 = 0 2. WRITE1 = 0 and WRITE2 = 0 3. WRITE0 = 0 and WRITE2 = 0	Wait for the write signals of the two processors in agreement to return to 0
FSM_9a	FSM_ERR1	TIMEOUT	The two processors in agreement did not return their write signals

Table 2 – *Continued on next page*

Table 2 – *Continued from previous page*

Current State	Next State	Transition Condition	Description
			to 0 in a timely fashion
FSM_10	FSM_11	MEM_READY = 1	Write data to memory and wait for memory to be ready
FSM_11	FSM_12	WRITE0 = 0 and WRITE1 = 0 and WRITE2 = 0 WRITE2 = 0	Pass MEM_READY signal back to all three processors and wait for write signals to return to 0
FSM_11	FSM_ERR1	TIMEOUT	At least one processor failed to receive the MEM_READY signal
FSM_12	FSM_0	MEM_READY = 0	Wait for memory ready signal to return to 0. End of store word FSM states
FSM_ERR0	FSM_0	ERR0 State = ERR0_14	ERR0 successfully resolved
FSM_ERR0	FSM_ERR1	ERR0 State = ERR0_ERR1	ERR0 encountered ERR1
FSM_ERR1	FSM_0	ERR1 State = ERR1_10	ERR1 successfully resolved
FSM_SAVE	FSM_0	SAVE State = SAVE_12	Save/restore point creation complete
FSM_SAVE	FSM_ERR1	SAVE State = SAVE_ERR1	SAVE encountered ERR1

Table 3. Normal Voter Outputs

Current State	Outputs
FSM_0	MEM_READ = 0, MEM_WRITE = 0 MEM_ADDRESS = 0, MEM_DATA_IN = 0 MEM_READY0 = 0, MEM_READY1 = 0, MEM_READY2 = 0 RESET0 = 0, RESET1 = 0, RESET2 = 0
FSM_1	
FSM_2	MEM_READ = READ0, MEM_ADDRESS = ADDRESS0
FSM_3	MEM_READ = 0, MEM_ADDRESS = 0 MEM_OUT0 = MEM_DATA_OUT, MEM_OUT1 = MEM_DATA_OUT, MEM_OUT2 = MEM_DATA_OUT MEM_READY0 = 1, MEM_READY1 = 1, MEM_READY2 = 1
FSM_3a	MEM_READ = 0, MEM_ADDRESS = 0 MEM_OUT0 = MEM_DATA_OUT, MEM_OUT1 = MEM_DATA_OUT, MEM_OUT2 = MEM_DATA_OUT MEM_READY0 = 0, MEM_READY1 = 0, MEM_READY2 = 0
FSM_4	MEM_READY0 = 0, MEM_READY1 = 0, MEM_READY2 = 0
FSM_5	
FSM_5a	1. MEM_READY0 = 1, MEM_READY1 = 1 2. MEM_READY1 = 1, MEM_READY2 = 1 3. MEM_READY0 = 1, MEM_READY2 = 1
FSM_6	MEM_READ = READ0, MEM_ADDRESS = ADDRESS0
FSM_7	MEM_READ = 0, MEM_ADDRESS = 0 MEM_OUT0 = MEM_DATA_OUT, MEM_OUT1 = MEM_DATA_OUT, MEM_OUT2 = MEM_DATA_OUT MEM_READY0 = 1, MEM_READY1 = 1, MEM_READY2 = 1
FSM_7a	MEM_READ = 0, MEM_ADDRESS = 0 MEM_OUT0 = MEM_DATA_OUT, MEM_OUT1 = MEM_DATA_OUT, MEM_OUT2 = MEM_DATA_OUT MEM_READY0 = 0, MEM_READY1 = 0, MEM_READY2 = 0
FSM_8	MEM_READY0 = 0, MEM_READY1 = 0, MEM_READY2 = 0

Table 3 – Continued on next page

Table 3 – *Continued from previous page*

Current State	Outputs
FSM_9	
FSM_9a	1. MEM_READY0 = 1, MEM_READY1 = 1 2. MEM_READY1 = 1, MEM_READY2 = 1 3. MEM_READY0 = 1, MEM_READY2 = 1
FSM_10	MEM_WRITE = WRITE0, MEM_ADDRESS = ADDRESS0, MEM_DATA_IN = DATA0
FSM_11	MEM_WRITE = 0, MEM_ADDRESS = 0 MEM_READY0 = 1, MEM_READY1 = 1, MEM_READY2 = 1
FSM_12	MEM_WRITE = 0, MEM_ADDRESS = 0 MEM_READY0 = 0, MEM_READY1 = 0, MEM_READY2 = 0

2.2 ERR0 TMR Voter Operation

ERR0 operation commences when the FSM enters state FSM_ERR0. During ERR0 operation, the voter proceeds through states ERR0_0 through ERR0_14. The states and their operation are illustrated in Table 4. Note that the COUNT starts at 0 and is incremented by 1 in ERR0_9 until it reaches a value of 30. COUNT is a 5 bit value. This ensures that all 31 registers are copied from the processors that agree to the incorrect processor.

Table 5 shows the outputs of the finite state machine when operating in ERR0 mode. If any of the voter output signals from Table 1 do not appear in the outputs for a particular state, then the output remains unchanged from the previous state in which the output was changed. Note that the CORRECT_BRANCH_DISTANCE is the distance the program counters on the correct processors need to jump to return to the point in the program at which the type ERR0 error was encountered. Similarly, the INCORRECT_BRANCH_DISTANCE is the distance the program counter on the incorrect processor needs to jump to return to the point in the program at which the type ERR0 error was encountered.

Table 4. ERR0 Voter Operation

Current State	Next State	Transition Condition	Description
ERR0_0	ERR0_1	FSM State = FSM_ERR0	ERR0 encountered by FSM
ERR0_1	ERR0_2	None	Reset the processor that is in error
ERR0_2	ERR0_3	1. (READ0 = 1 and READ1 = 1 and ADDRESS0 = ADDRESS1) 2. (READ1 = 1 and READ2 = 1 and ADDRESS1 = ADDRESS2) 3. (READ0 = 1 and READ2 = 1 and ADDRESS0 = ADDRESS2)	Wait for correct processors to attempt to read an instruction from memory
ERR0_2	ERR0_ERR1	1. READ0 = 1 and READ1 = 1 2. READ1 = 1 and READ2 = 1 3. READ0 = 1 and READ2 = 1	The two processors in agreement did not have the same address
ERR0_2	ERR0_ERR1	TIMEOUT	One or both processors in agreement did not attempt to read an instruction in a timely fashion
ERR0_3	ERR0_4	1. READ0 = 0 and READ1 = 0 2. READ1 = 0 and READ2 = 0 3. READ0 = 0 and READ2 = 0	Issue store word commands to the processors in agreement
ERR0_3	ERR0_ERR1	TIMEOUT	One or both processors in agreement did not read the instruction in a timely fashion
ERR0_4	ERR0_5	1. WRITE0 = 1 and WRITE1 = 1 2. WRITE1 = 1 and WRITE2 = 1 3. WRITE0 = 1 and WRITE2 = 1	Wait for the processors in agreement to write a word
ERR0_4	ERR0_ERR1	TIMEOUT	One or both processors in agreement did not attempt to store a word

Table 4 – Continued on next page

Table 4 – *Continued from previous page*

Current State	Next State	Transition Condition	Description
			in a timely fashion
ERR0_5	ERR0_6	1. WRITE0 = 0 and WRITE1 = 0 and DATA0 = DATA1 2. WRITE0 = 1 and WRITE2 = 0 and DATA1 = DATA2 3. WRITE0 = 0 and WRITE2 = 0 and DATA0 = DATA2	Check that the data from the processors in agreement provided the same data and wait for write signals to return to 0
ERR0_5	ERR0_ERR1	1. WRITE0 = 0 and WRITE1 = 0 and DATA0 != DATA1 2. WRITE0 = 1 and WRITE2 = 0 and DATA1 != DATA2 3. WRITE0 = 0 and WRITE2 = 0 and DATA0 != DATA2	Data from the processors in agreement provided different data
ERR0_5	ERR0_ERR1	TIMEOUT	The processors in agreement did not return their write signals to 0 in a timely fashion
ERR0_6	ERR0_7	1. READ2 = 1 2. READ0 = 1 3. READ1 = 1	Wait for incorrect processor to attempt to read an instruction
ERR0_6	ERR0_ERR1	TIMEOUT	The incorrect processor did not attempt to read an instruction in a timely fashion
ERR0_7	ERR0_8	1. READ2 = 0 2. READ0 = 0 3. READ1 = 0	Issue load word command to incorrect processor
ERR0_7	ERR0_ERR1	TIMEOUT	The incorrect processor did not

Table 4 – *Continued on next page*

Table 4 – *Continued from previous page*

Current State	Next State	Transition Condition	Description
			receive the load word instruction in a timely fashion
ERR0_8	ERR0_9	1. READ2 = 1 2. READ0 = 1 3. READ1 = 1	Wait for incorrect processor to attempt to load a word
ERR0_8	ERR0_ERR1	TIMEOUT	The incorrect processor did not attempt to load a word in a timely fashion
ERR0_9	ERR0_2	1. READ2 = 0 2. READ0 = 0 3. READ1 = 0 and COUNT < 30	Return requested data to incorrect processor. Return to ERR0_2 to copy next register.
ERR0_9	ERR0_10	1. READ2 = 0 2. READ0 = 0 3. READ1 = 0 and COUNT $\geq$ 30	Return requested data to incorrect processor. All registers have been copied.
ERR0_9	ERR0_ERR1	TIMEOUT	The incorrect processor did not receive the requested data in a timely fashion
ERR0_10	ERR0_11	1. (READ0 = 1 and READ1 = 1 and ADDRESS0 = ADDRESS1) 2. (READ1 = 1 and READ2 = 1 and ADDRESS1 = ADDRESS2) 3. (READ0 = 1 and READ2 = 1 and ADDRESS0 = ADDRESS2)	Wait for correct processors to attempt to read an instruction from memory
ERR0_10	ERR0_ERR1	1. READ0 = 1 and READ1 = 1	The two processors in agreement

Table 4 – *Continued on next page*

Table 4 – *Continued from previous page*

Current State	Next State	Transition Condition	Description
		2. READ1 = 1 and READ2 = 1 3. READ0 = 1 and READ2 = 1	did not have the same address
ERR0_10	ERR0_ERR1	TIMEOUT	One or both processors in agreement did not attempt to read an instruction in a timely fashion
ERR0_11	ERR0_12	1. READ0 = 0 and READ1 = 0 2. READ1 = 0 and READ2 = 0 3. READ0 = 0 and READ2 = 0	Issue branch commands to the processors in agreement. Wait for read signals to return to 0
ERR0_11	ERR0_ERR1	TIMEOUT	One or both processors in agreement did not read the instruction in a timely fashion
ERR0_12	ERR0_13	1. READ2 = 1 2. READ0 = 1 3. READ1 = 1	Wait for incorrect processor to attempt to read an instruction
ERR0_12	ERR0_ERR1	TIMEOUT	The incorrect processor did not attempt to read an instruction in a timely fashion
ERR0_13	ERR0_14	1. READ2 = 0 2. READ0 = 0 3. READ1 = 0	Issue branch command to incorrect processor. Wait for read signal to return to 0
ERR0_13	ERR0_ERR1	TIMEOUT	The incorrect processor did not receive the branch instruction in a timely fashion
ERR0_14	ERR0_0	FSM State != FSM_ERR0	Wait for normal operation to resume
ERR0_ERR1	ERR0_0	FSM State != FSM_ERR0	Wait for FSM_ERR1 operation to start

Table 5. ERR0 Voter Outputs

Current State	Outputs
ERR0_0	READY0 = 0, READY1 = 0, READY2 = 0
ERR0_1	MEM_READY0 = 0, MEM_READY1 = 0, MEM_READY2 = 0 1. RESET2 = 1 2. RESET0 = 1 3. RESET1 = 1
ERR0_2	MEM_READY0 = 0, MEM_READY1 = 0, MEM_READY2 = 0 1. RESET2 = 0 2. RESET0 = 0 3. RESET1 = 0
ERR0_3	1. MEM_OUT0 = '10101100000' & (COUNT+1) & '0000000000000000' MEM_OUT1 = '10101100000' & (COUNT+1) & '0000000000000000' MEM_READY0 = 1, MEM_READY1 = 1 2. MEM_OUT1 = '10101100000' & (COUNT+1) & '0000000000000000' MEM_OUT2 = '10101100000' & (COUNT+1) & '0000000000000000' MEM_READY1 = 1, MEM_READY2 = 1 3. MEM_OUT0 = '10101100000' & (COUNT+1) & '0000000000000000' MEM_OUT2 = '10101100000' & (COUNT+1) & '0000000000000000' MEM_READY0 = 1, MEM_READY2 = 1
ERR0_4	1. MEM_READY0 = 0, MEM_READY1 = 0 2. MEM_READY1 = 0, MEM_READY2 = 0 3. MEM_READY0 = 0, MEM_READY2 = 0
ERR0_5	1. MEM_READY0 = 1, MEM_READY1 = 1 2. MEM_READY1 = 1, MEM_READY2 = 1 3. MEM_READY0 = 1, MEM_READY2 = 1
ERR0_6	1. MEM_READY0 = 0, MEM_READY1 = 0

Table 5 – Continued on next page

Table 5 – *Continued from previous page*

Current State	Outputs
	2. MEM_READY1 = 0, MEM_READY2 = 0 3. MEM_READY0 = 0, MEM_READY2 = 0
ERR0_7	1. MEM_OUT2 = '10001100000' & (COUNT+1) & '00000000000000000' MEM_READY2 = 1 2. MEM_OUT0 = '10001100000' & (COUNT+1) & '00000000000000000' MEM_READY0 = 1 3. MEM_OUT1 = '10001100000' & (COUNT+1) & '00000000000000000' MEM_READY1 = 1
ERR0_8	1. MEM_READY2 = 0 2. MEM_READY0 = 0 3. MEM_READY1 = 0
ERR0_9	1. MEM_OUT2 = DATA0, MEM_READY2 = 1 2. MEM_OUT0 = DATA1, MEM_READY0 = 1 3. MEM_OUT1 = DATA0, MEM_READY1 = 1
ERR0_10	1. MEM_READY2 = 0 2. MEM_READY0 = 0 3. MEM_READY1 = 0
ERR0_11	1. MEM_OUT0 = '0001000000000000' & CORRECT_BRANCH_DISTANCE MEM_OUT1 = '0001000000000000' & CORRECT_BRANCH_DISTANCE MEM_READY0 = 1, MEM_READY1 = 1 2. MEM_OUT1 = '0001000000000000' & CORRECT_BRANCH_DISTANCE MEM_OUT2 = '0001000000000000' & CORRECT_BRANCH_DISTANCE MEM_READY1 = 1, MEM_READY2 = 1 3. MEM_OUT0 = '0001000000000000' & CORRECT_BRANCH_DISTANCE MEM_OUT2 = '0001000000000000' & CORRECT_BRANCH_DISTANCE

Table 5 – *Continued on next page*

Table 5 – *Continued from previous page*

Current State	Outputs
	MEM_READY0 = 1, MEM_READY2 = 1
ERR0_12	1. MEM_READY0 = 0, MEM_READY1 = 0 2. MEM_READY1 = 0, MEM_READY2 = 0 3. MEM_READY0 = 0, MEM_READY2 = 0
ERR0_13	1. MEM_OUT2 = '0001000000000000' & INCORRECT_BRANCH_DISTANCE MEM_READY2 = 1 2. MEM_OUT0 = '0001000000000000' & INCORRECT_BRANCH_DISTANCE MEM_READY0 = 1 3. MEM_OUT1 = '0001000000000000' & INCORRECT_BRANCH_DISTANCE MEM_READY1 = 1
ERR0_14	1. MEM_READY2 = 0 2. MEM_READY0 = 0 3. MEM_READY1 = 0

2.3 ERR1 TMR Voter Operation

ERR1 operation commences when the FSM enters state FSM_ERR1. During ERR1 operation, the voter proceeds through states ERR1_0 through ERR1_10. The states and their operation are illustrated in Table 6. Note that the COUNT starts at 0 and is incremented by 1 in ERR1_6 until it reaches a value of 30. COUNT is a 5 bit value. This ensures that all 31 registers are copied from the save/restore point to the processors.

Table 7 shows the outputs of the finite state machine when operating in ERR1 mode. If any of the voter output signals from Table1 do not appear in the outputs for a particular state, then the output remains unchanged from the previous state in which the output was changed. Note that the MEM_START is the memory location for the save/restore point memory. Also note that the BRANCH_DISTANCE is the distance the program counter on the processors need to jump to return to the point in the program at which the active save/restore point was created.

Table 6. ERR1 Voter Operation

Current State	Next State	Transition Condition	Description
ERR1_0	ERR1_1	FSM State = FSM_ERR1	ERR1 encountered by FSM
ERR1_1	ERR1_2	MEM_READY = 1	Reset all three processors and wait for save/restore point location to load from memory
ERR1_2	ERR1_2a	MEM_READY = 0	Wait for memory ready signal to return to 0
ERR1_2a	ERR1_3	READ0 = 1 and READ1 = 1 and READ2 = 1 and ADDRESS0 = ADDRESS1 and ADDRESS0 = ADDRESS2	Wait for all three processors to attempt to read an instruction
ERR1_2a	ERR1_0	READ0 = 1 and READ1 = 1 READ2 = 1 and (ADDRESS0 != ADDRESS1 or ADDRESS1 != ADDRESS2 or ADDRESS0 != ADDRESS2)	All three processors attempted to read an instruction but from different addresses
ERR1_2a	ERR1_0	TIMEOUT	One or more processors did not attempt to read an instruction in a timely fashion
ERR1_3	ERR1_4	READ0 = 0 and READ1 = 0 READ2 = 0	Issue a load command to all three processors to load a register value
ERR1_3	ERR1_0	TIMEOUT	One or more processors did not receive the load command in a timely fashion
ERR1_4	ERR1_5	READ0 = 1 and READ1 = 1 READ2 = 1 and ADDRESS0 = ADDRESS1 and READ2 = 1 and ADDRESS0 = ADDRESS2	Wait for all three processors to attempt to load data

Table 6 – Continued on next page

Table 6 – *Continued from previous page*

Current State	Next State	Transition Condition	Description
ERR1_4	ERR1_0	READ0 = 1 and READ1 = 1 READ2 = 1 and (ADDRESS0 != ADDRESS1 or ADDRESS1 != ADDRESS2 or ADDRESS0 != ADDRESS2)	All three processors attempted to load data but from different addresses
ERR1_4	ERR1_0	TIMEOUT	One or more processors did not attempt to load data in a timely fashion
ERR1_5	ERR1_6	MEM_READY = 1	Read data from memory Wait for memory to be ready
ERR1_6	ERR1_2	READ0 = 0 and READ1 = 0 READ2 = 0 and COUNT < 30	Return data from memory to all three processors. Return to ERR1_2 to copy next register
ERR1_6	ERR1_6a	READ0 = 0 and READ1 = 0 READ2 = 0 and COUNT ≥ 30	Return data from memory to all three processors. All registers have been copied.
ERR1_6	ERR1_0	TIMEOUT	One or more processors did not receive the data from memory
ERR1_6a	ERR1_7	MEM_READY = 0	Wait for memory ready signal to return to 0
ERR1_7	ERR1_8	MEM_READY = 1	Load desired program location from memory
ERR1_8	ERR1_8a	MEM_READY = 0	Wait for memory ready signal to return to 0
ERR1_8a	ERR1_9	READ0 = 1 and READ1 = 1 and READ2 = 1 and ADDRESS0 = ADDRESS1 and	Wait for all three processors to attempt to read an instruction

Table 6 – *Continued on next page*

Table 6 – *Continued from previous page*

Current State	Next State	Transition Condition	Description
		ADDRESS0 = ADDRESS2	
ERR1_8a	ERR1_0	READ0 = 1 and READ1 = 1 READ2 = 1 and (ADDRESS0 != ADDRESS1 or ADDRESS1 != ADDRESS2 or ADDRESS0 != ADDRESS2)	All three processors attempted to read an instruction but from different addresses
ERR1_8a	ERR1_0	TIMEOUT	One or more processors did not attempt to read an instruction in a timely fashion
ERR1_9	ERR1_10	READ0 = 0 and READ1 = 0 READ2 = 0	Issue a branch command to all three processors to return to the program location
ERR1_9	ERR1_0	TIMEOUT	One or more processors did not receive the branch command in a timely fashion
ERR1_10	ERR1_0	FSM State != FSM_ERR1	Wait for normal operation to resume

Table 7. ERR1 Voter Outputs

Current State	Outputs
ERR1_0	READY0 = 0, READY1 = 0, READY2 = 0
ERR1_1	RESET0 = 1, RESET1 = 1, RESET2 = 1 READY0 = 0, READY1 = 0, READY2 = 0 MEM_READ = 1, MEM_ADDRESS = MEM_START + 4 × 64
ERR1_2	RESET0 = 0, RESET1 = 0, RESET2 = 0 READY0 = 0, READY1 = 0, READY2 = 0 MEM_READ = 0, MEM_ADDRESS = 0
ERR1_2a	RESET0 = 0, RESET1 = 0, RESET2 = 0 READY0 = 0, READY1 = 0, READY2 = 0 MEM_READ = 0, MEM_ADDRESS = 0
ERR1_3	READY0 = 1, READY1 = 1, READY2 = 1 If active save/restore point is 0 MEM_OUT0 = '10001100000' & (COUNT+1) & (MEM_START + 4 × COUNT) MEM_OUT1 = '10001100000' & (COUNT+1) & (MEM_START + 4 × COUNT) MEM_OUT2 = '10001100000' & (COUNT+1) & (MEM_START + 4 × COUNT) If active save/restore point is 1 MEM_OUT0 = '10001100000' & (COUNT+1) & (MEM_START + 4 × 32 + 4 × COUNT) MEM_OUT1 = '10001100000' & (COUNT+1) & (MEM_START + 4 × 32 + 4 × COUNT) MEM_OUT2 = '10001100000' & (COUNT+1) & (MEM_START + 4 × 32 + 4 × COUNT)
ERR1_4	READY0 = 0, READY1 = 0, READY2 = 0
ERR1_5	MEM_READ = 1, MEM_ADDRESS = ADDRESS0
ERR1_6	MEM_READ = 0, MEM_ADDRESS = 0 READY0 = 1, READY1 = 1, READY2 = 1 MEM_OUT0 = MEM_DATA_OUT, MEM_OUT1 = MEM_DATA_OUT, MEM_OUT2 = MEM_DATA_OUT
ERR1_6a	MEM_READ = 0, MEM_ADDRESS = 0

Table 7 – Continued on next page

Table 7 – *Continued from previous page*

Current State	Outputs
	READY0 = 0, READY1 = 0, READY2 = 0 MEM_OUT0 = MEM_DATA_OUT, MEM_OUT1 = MEM_DATA_OUT, MEM_OUT2 = MEM_DATA_OUT
ERR1_7	READY0 = 0, READY1 = 0, READY2 = 0 MEM_READ = 1 If active save/restore point is 0 MEM_ADDRESS = MEM_START + 4 × 31 If active save/restore point is 1 MEM_ADDRESS = MEM_START + 4 × 63
ERR1_8	MEM_READ = 0, MEM_ADDRESS = 0
ERR1_8a	MEM_READ = 0, MEM_ADDRESS = 0
ERR1_9	READY0 = 1, READY1 = 1, READY2 = 1 MEM_OUT0 = '0001000000000000' & BRANCH_DISTANCE MEM_OUT1 = '0001000000000000' & BRANCH_DISTANCE MEM_OUT2 = '0001000000000000' & BRANCH_DISTANCE
ERR1_10	READY0 = 0, READY1 = 0, READY2 = 0

2.4 SAVE TMR Voter Operation

During SAVE operation, the voter proceeds through states SAVE_0 through SAVE_12. The states and their operation are illustrated in Table 8. Note that the COUNT starts at 0 and is incremented by 1 in SAVE_6 until it reaches a value of 30. COUNT is a 5 bit value. This ensures that all 31 registers are copied from the processors to the save/restore point.

Table 9 shows the outputs of the finite state machine when operating in SAVE mode. If any of the voter output signals from Table1 do not appear in the outputs for a particular state, then the output remains unchanged from the previous state in which the output was changed. Note that the MEM_START is the memory location for the save/restore point memory. Also note that the BRANCH_DISTANCE is the distance the program counter on the processors need to jump to return to the point in the program at which save/restore point creation began. Finally, PC_SAVE is the program counter value at which the save/restore point creation process began.

Table 8. SAVE Voter Operation

Current State	Next State	Transition Condition	Description
SAVE_0	SAVE_1	FSM State = FSM_SAVE	SAVE point encountered by FSM
SAVE_1	SAVE_2	MEM_READY = 1	Load save/restore point from memory
SAVE_2	SAVE_2a	MEM_READY = 0	Wait for memory ready signal to return to 0
SAVE_2a	SAVE_3	READ0 = 1 and READ1 = 1 and READ2 = 1 and ADDRESS0 = ADDRESS1 and ADDRESS0 = ADDRESS2	Wait for all three processors to attempt to read an instruction
SAVE_2a	SAVE_ERR1	READ0 = 1 and READ1 = 1 and READ2 = 1 and (ADDRESS0 != ADDRESS1 or ADDRESS1 != ADDRESS2 or ADDRESS0 != ADDRESS2)	All three processors attempted to read an instruction but from different addresses
SAVE_2a	SAVE_ERR1	TIMEOUT	One or more processors did not attempt to read an instruction in a timely fashion
SAVE_3	SAVE_4	READ0 = 0 and READ1 = 0 and READ2 = 0	Issue a store command to all three processors to store a register value to memory
SAVE_3	SAVE_ERR1	TIMEOUT	One or more processors did not receive the store command in a timely fashion
SAVE_4	SAVE_5	WRITE0 = 1 and WRITE1 = 1 and WRITE2 = 1 and (DATA0 = DATA1 or	Wait for all three processors to attempt to store the register value to memory. At least two

Table 8 – Continued on next page

Table 8 – Continued from previous page

Current State	Next State	Transition Condition	Description
		DATA1 = DATA2 or DATA0 = DATA2)	of three register values must agree
SAVE_4	SAVE_ERR1	WRITE0 = 1 and WRITE1 = 1 and WRITE2 = 1 and (DATA0 != DATA1 and DATA1 != DATA2 and and DATA0 != DATA2)	No two register values agreed
SAVE_4	SAVE_ERR1	TIMEOUT	One or more processors did not attempt to store a word in a timely fashion
SAVE_5	SAVE_6	MEM_READY = 1	Save register data to memory Wait for memory to be ready
SAVE_6	SAVE_2	WRITE0 = 0 and WRITE1 = 0 and WRITE2 = 0 and COUNT < 30	Wait for processor write signals to return to zero. Return to SAVE_2 to save the next register
SAVE_6	SAVE_6a	WRITE0 = 0 and WRITE1 = 0 and WRITE2 = 0 and COUNT $\geq$ 30	Wait for processor write signals to return to zero. All registers have been saved
SAVE_6	SAVE_ERR1	TIMEOUT	One or more processors did not receive the memory ready response in a timely fashion
SAVE_6a	SAVE_7	MEM_READY = 0	Wait for MEM_READY to return to zero
SAVE_7	SAVE_8	MEM_READY = 1	Write updated PC address at which save/restore point was

Table 8 – Continued on next page

Table 8 – *Continued from previous page*

Current State	Next State	Transition Condition	Description
			created to memory
SAVE_8	SAVE_9	MEM_READY = 0	Wait for MEM_READY to return to zero
SAVE_9	SAVE_10	MEM_READY = 1	Update the location of the save/restore point in memory
SAVE_10	SAVE_10a	MEM_READY = 0	Wait for MEM_READY to return to zero
SAVE_10a	SAVE_11	READ0 = 1 and READ1 = 1 and READ2 = 1 and ADDRESS0 = ADDRESS1 and ADDRESS0 = ADDRESS2	Wait for all three processors to attempt to read an instruction
SAVE_10a	SAVE_ERR1	READ0 = 1 and READ1 = 1 READ2 = 1 and (ADDRESS0 != ADDRESS1 or ADDRESS1 != ADDRESS2 or ADDRESS0 != ADDRESS2)	All three processors attempted to read an instruction but from different addresses
SAVE_10a	SAVE_ERR1	TIMEOUT	One or more processors did not attempt to read an instruction in a timely fashion
SAVE_11	SAVE_12	READ0 = 0 and READ1 = 0 READ2 = 0	Issue a branch command to all three processors to return to the program location at which the save creation point started
SAVE_11	SAVE_ERR1	TIMEOUT	One or more processors did not receive the branch command

Table 8 – *Continued on next page*

Table 8 – *Continued from previous page*

Current State	Next State	Transition Condition	Description
			in a timely fashion
SAVE_12	SAVE_0	FSM State != FSM_SAVE	Wait for normal operation to resume
SAVE_ERR1	SAVE_0	FSM State != FSM_SAVE	Wait for FSM_ERR1 operation to start

Table 9. SAVE Voter Outputs

Current State	Outputs
SAVE_0	READY0 = 0, READY1 = 0, READY2 = 0
SAVE_1	MEM_READ = 1, MEM_ADDRESS = MEM_START + 4 × 64
SAVE_2	MEM_READ = 0, MEM_ADDRESS = 0 READY0 = 0, READY1 = 0, READY2 = 0
SAVE_2a	MEM_READ = 0, MEM_ADDRESS = 0 READY0 = 0, READY1 = 0, READY2 = 0
SAVE_3	READY0 = 1, READY1 = 1, READY2 = 1 If active save/restore point is 0 MEM_OUT0 = '10101100000' & (COUNT+1) & (MEM_START + 4 × COUNT) MEM_OUT1 = '10101100000' & (COUNT+1) & (MEM_START + 4 × COUNT) MEM_OUT2 = '10101100000' & (COUNT+1) & (MEM_START + 4 × COUNT) If active save/restore point is 1 MEM_OUT0 = '10101100000' & (COUNT+1) & (MEM_START + 4 × 32 + 4 × COUNT) MEM_OUT1 = '10101100000' & (COUNT+1) & (MEM_START + 4 × 32 + 4 × COUNT) MEM_OUT2 = '10101100000' & (COUNT+1) & (MEM_START + 4 × 32 + 4 × COUNT)
SAVE_4	READY0 = 0, READY1 = 0, READY2 = 0
SAVE_5	MEM_WRITE = 1 If active save/restore point is 1 MEM_ADDRESS = '0000000000000000' & (MEM_START + 4 × COUNT) If active save/restore point is 0 MEM_ADDRESS = '0000000000000000' & (MEM_START + 4 × 32 + 4 × COUNT) If DATA0 = DATA1 then MEM_DATA_IN = DATA0 If DATA1 = DATA2 then MEM_DATA_IN = DATA1 If DATA01 = DATA2 then MEM_DATA_IN = DATA0
SAVE_6	MEM_WRITE = 0, MEM_ADDRESS = 0, MEM_DATA_IN = 0

Table 9 – Continued on next page

Table 9 – *Continued from previous page*

Current State	Outputs
	READY0 = 1, READY1 = 1, READY2 = 1
SAVE_6a	MEM_WRITE = 0, MEM_ADDRESS = 0, MEM_DATA_IN = 0 READY0 = 0, READY1 = 0, READY2 = 0
SAVE_7	MEM_WRITE = 1 If active save/restore point is 1 MEM_ADDRESS = '0000000000000000' & (MEM_START + 4 × 31 If active save/restore point is 0 MEM_ADDRESS = '0000000000000000' & (MEM_START + 4 × 63 MEM_DATA_IN = PC_SAVE READY0 = 0, READY1 = 0, READY2 = 0
SAVE_8	MEM_WRITE = 0, MEM_ADDRESS = 0, MEM_DATA_IN = 0
SAVE_9	MEM_WRITE = 1 MEM_ADDRESS = MEM_START + 4 × 64 If active save/restore point is 1 MEM_DATA_IN = 0 If active save/restore point is 0 MEM_DATA_IN = 1
SAVE_10	MEM_WRITE = 0, MEM_ADDRESS = 0, MEM_DATA_IN = 0
SAVE_10a	MEM_WRITE = 0, MEM_ADDRESS = 0, MEM_DATA_IN = 0
SAVE_11	READY0 = 1, READY1 = 1, READY2 = 1 MEM_OUT0 = '0001000000000000' & BRANCH_DISTANCE MEM_OUT1 = '0001000000000000' & BRANCH_DISTANCE MEM_OUT2 = '0001000000000000' & BRANCH_DISTANCE
SAVE_12	READY0 = 0, READY1 = 0, READY2 = 0

Appendix A. TMR Voter VHDL Code

This appendix provides an example of the TMR Voter. This voter has been customized so so that it performs save/restore point creation once after every 10,000 instructions are processed. This value is set by the *k_save_point* variable. It has also been customized to point to a specific memory location for the save/restore point. This value is set by the *k_mem_location* variable.

```
1  —| TMR_Voter_Test1001.vhd
   —| Author: Nicolas Hamilton using writeTMR_VOTER2.m
3  —| Created: 23 July 2019 at 20:19:02
   —| Test the functionality of the Basic_MIPS using the
      MEMEMULATOR
5  —| so that the controller can "read" instructions from
      memory, transition
   —| through the appropriate states, and send the proper
      control signals out
7  library IEEE;
   use IEEE.std_logic_1164.all;
9  use IEEE.numeric_std.all;

11 entity TMR_Voter_Test1001 is
    port (i_clk           : in  std_logic;
13         i_reset        : in  std_logic;
         i_MEM_READ0      : in  std_logic;
15         i_MEM_READ1     : in  std_logic;
         i_MEM_READ2      : in  std_logic;
17         i_MEM_WRITE0    : in  std_logic;
         i_MEM_WRITE1     : in  std_logic;
19         i_MEM_WRITE2    : in  std_logic;
         i_MEM_IN0        : in  std_logic_vector(31 downto 0);
21         i_MEM_IN1       : in  std_logic_vector(31 downto 0);
         i_MEM_IN2        : in  std_logic_vector(31 downto 0);
23         i_MEM_ADDRESS0  : in  std_logic_vector(31 downto 0)
        ;
         i_MEM_ADDRESS1   : in  std_logic_vector(31 downto 0)
        ;
25         i_MEM_ADDRESS2  : in  std_logic_vector(31 downto 0)
        ;
         i_MEM_OUT         : in  std_logic_vector(31 downto 0);
27         i_MEM_READY     : in  std_logic;
```

```

29         o_MEM_READ      : out std_logic;
        o_MEM_WRITE      : out std_logic;
        o_MEM_IN         : out std_logic_vector(31 downto 0)
;
31         o_MEM_ADDRESS   : out std_logic_vector(31 downto 0);
        o_MEM_OUT0        : out std_logic_vector(31 downto 0);
33         o_MEM_OUT1      : out std_logic_vector(31 downto 0);
        o_MEM_OUT2        : out std_logic_vector(31 downto 0);
35         o_MEM_READY0    : out std_logic;
        o_MEM_READY1      : out std_logic;
37         o_MEM_READY2    : out std_logic;
        o_RESET0          : out std_logic;
39         o_RESET1        : out std_logic;
        o_RESET2          : out std_logic;
41         o_NEXT_INSTR    : out std_logic;
        o_TMR_ERROR       : out std_logic;
43         o_Err_Override   : out std_logic);
end TMR_Voter_Test1001;
45
architecture a_TMR_Voter_Test1001 of TMR_Voter_Test1001 is
47 —| Declare components
—| Create state machine types
49 — Create states for the voter finite state machine
    type sm_vfsm is (s_vfsm_0 , s_vfsm_1 , s_vfsm_2 , s_vfsm_3 ,
51                     s_vfsm_3a , s_vfsm_4 , s_vfsm_5 , s_vfsm_5a ,
                     s_vfsm_6 , s_vfsm_7 , s_vfsm_7a , s_vfsm_8 ,
53                     s_vfsm_8a , s_vfsm_9 , s_vfsm_9a , s_vfsm_10 ,
                     s_vfsm_11 , s_vfsm_12 ,
55                     s_vfsm_err0 , s_vfsm_err1 , s_vfsm_save);
    — Create states for the voter err0 finite state machine
57    type sm_verr0 is (s_verr0_0 , s_verr0_1 , s_verr0_2 ,
                      s_verr0_3 ,
                      s_verr0_4 , s_verr0_5 , s_verr0_6 ,
                      s_verr0_7 ,
59                      s_verr0_8 , s_verr0_9 , s_verr0_10 ,
                      s_verr0_11 ,
                      s_verr0_12 , s_verr0_13 , s_verr0_14 ,
                      s_verr0_err1);
61    — Create states for the voter err1 finite state machine
    type sm_verr1 is (s_verr1_0 , s_verr1_1 , s_verr1_2 ,
                      s_verr1_2a ,
63                      s_verr1_3 , s_verr1_4 , s_verr1_5 ,
                      s_verr1_6 ,

```

```

65         s_verr1_6a , s_verr1_7 , s_verr1_8 ,
s_verr1_8a ,
67         s_verr1_9 , s_verr1_10 , s_verr1_11 ,
s_verr1_12 ,
        s_verr1_13 , s_verr1_14 );
67 — Create states for the voter sav finite state machine
type sm_vsav is (s_vsav_0 , s_vsav_1 , s_vsav_2 ,
s_vsav_2a ,
69         s_vsav_3 , s_vsav_4 , s_vsav_5 ,
s_vsav_6 ,
        s_vsav_6a , s_vsav_7 , s_vsav_8 ,
s_vsav_9 ,
71         s_vsav_10 , s_vsav_10a , s_vsav_11 ,
s_vsav_12 , s_vsav_err1 );
73
75 — Initialize the voter finite state machine register
signal f_vfsm_state : sm_vfsm := s_vfsm_0;
77 — Initialize the err0 finite state machine register
signal f_verr0_state : sm_verr0 := s_verr0_0;
79 — Initialize the err0 finite state machine register
signal f_verr1_state : sm_verr1 := s_verr1_0;
81 — Initialize the save finite state machine register
signal f_vsav_state : sm_vsav := s_vsav_0;

83 —| Define Signals
85 — Counter used to determine when to create a save point
signal f_instr_count : unsigned(31 downto 0) := (others =>
'0');
87 — Constant used to compare instruction counter against
constant k_save_point : unsigned(31 downto 0) := "
0000000000000000000010011100010000"; —10,000

89 — Singlas to determine which processor is wrong in the
s_vfsm_err0 state
signal f_wrong0 : std_logic := '0';
91 signal f_wrong1 : std_logic := '0';
signal f_wrong2 : std_logic := '0';
93 — Determine when to return from the s_vfsm_err0 state
signal f_done_wrong0 : std_logic := '0';
95 signal f_done_wrong1 : std_logic := '0';
signal f_done_wrong2 : std_logic := '0';
97 — Determine when to return from the s_vfsm_err1 state
signal f_done_all_wrong : std_logic := '0';

```

```

99  — Determine when to return from the s_vfsm_save state
    signal f_done_save : std_logic := '0';
101 — Count signals to determine which register to load in
    error correction processes
    signal f_err0_count : unsigned(4 downto 0) := (others =>
        '0');
103 signal f_err1_count : unsigned(4 downto 0) := (others =>
        '0');
    — Count signal to determine which register to save in
    save point processes
105 signal f_save_count : unsigned(4 downto 0) := (others =>
        '0');
    — Signal to temporarily store data to be transferred from
    memory to processors
107 — for non-erroneous uperation
    signal f_data : std_logic_vector(31 downto 0) := (others
=> '0');
109 — Signal to temporarily store data to be transferred from
    correct processors
    — to incorrect processor for type 0 errors
111 signal f_err0_data : std_logic_vector(31 downto 0) := (
    others => '0');
    — Signal to temporarily store data to be transferred from
    memory to processors
113 — for type 1 errors
    signal f_err1_data : std_logic_vector(31 downto 0) := (
    others => '0');
115 — Signal to store the address at which the error occurred
    signal f_err_address : std_logic_vector(31 downto 0) := (
    others => '0');
117 — Signal to store the address of the current instruction
    signal f_current_address : std_logic_vector(31 downto 0)
:= (others => '0');
119 — Signal to store the address of the current instruction
    signal f_save_current_address : std_logic_vector(31 downto
    0) := (others => '0');
121 — Signal to store the address of the instruction at which
    save point creation started
    signal f_save_address : std_logic_vector(31 downto 0) := (
    others => '0');
123 — Signals to compute the branch distance for type 0
    errors
    signal w_correct_branch_distance32 : std_logic_vector(31

```

```

125     downto 0);
    signal w_correct_branch_distance16 : std_logic_vector(15
    downto 0);
    signal w_incorrect_branch_distance32 : std_logic_vector(31
    downto 0);
127     signal w_incorrect_branch_distance16 : std_logic_vector(15
    downto 0);
    — Signals to compute the branch distance for type 1
    errors
129     signal w_err1_branch_distance32 : std_logic_vector(31
    downto 0);
    signal w_err1_branch_distance16 : std_logic_vector(15
    downto 0);
131     — Signals to compute the branch distance for returning
    after creating a save point
    signal w_save_branch_distance32 : std_logic_vector(31
    downto 0);
133     signal w_save_branch_distance16 : std_logic_vector(15
    downto 0);

    — Watchdog timers to detect a timeout error
    signal f_fsm_timer : unsigned(5 downto 0) := (others =>
    '0');
137     signal f_err0_timer : unsigned(5 downto 0) := (others =>
    '0');
    signal f_err1_timer : unsigned(5 downto 0) := (others =>
    '0');
139     signal f_save_timer : unsigned(5 downto 0) := (others =>
    '0');
    constant k_timeout : unsigned(5 downto 0) := "110010";
141     — Registers to keep track of memory save point (0 or 1)
    signal f_err1_save_point32 : std_logic_vector(31 downto 0)
    := (others => '0');
143     signal f_save_save_point32 : std_logic_vector(31 downto 0)
    := (others => '0');
    — Location at which backup memory location starts
145     constant k_mem_location : std_logic_vector(31 downto 0) :=
    "00000000000000000000000010111001000";
    — Constant used to access the 64th memory location (0–30=
    R1–R31, 31=PC, 32–62=R1–R31, 63=PC, 64=sav_point)
147     constant k_256_32 : std_logic_vector(31 downto 0) := "
    000000000000000000000000100000000";
    — Constant used to access the 32nd memory location (0–30=

```

```

149 R1-R31, 31=PC, 32-62=R1-R31, 63=PC, 64=sav_point)
constant k_128_16 : std_logic_vector(15 downto 0) := "
0000000010000000";
— Constant used to access the 31st memory location (0-30=
R1-R31, 31=PC, 32-62=R1-R31, 63=PC, 64=sav_point)
151 constant k_124_32 : std_logic_vector(31 downto 0) := "
00000000000000000000000001111100";
— Constant used to access the 63rd memory location (0-30=
R1-R31, 31=PC, 32-62=R1-R31, 63=PC, 64=sav_point)
153 constant k_252_32 : std_logic_vector(31 downto 0) := "
00000000000000000000000001111100";
— Constant used to retrieve the desired PC location for
type 1 error recovery
155 signal f_err1_pc : std_logic_vector(31 downto 0) := (
others => '0');

157 — Registers for holding output values
signal f_MEM_READ      : std_logic := '0';
159 signal f_MEM_WRITE   : std_logic := '0';
signal f_MEM_IN        : std_logic_vector(31 downto 0) :=
(others => '0');
161 signal f_MEM_ADDRESS : std_logic_vector(31 downto 0) :=
(others => '0');
signal f_MEM_OUT0      : std_logic_vector(31 downto 0) :=
(others => '0');
163 signal f_MEM_OUT1    : std_logic_vector(31 downto 0) :=
(others => '0');
signal f_MEM_OUT2      : std_logic_vector(31 downto 0) :=
(others => '0');
165 signal f_MEM_READY0  : std_logic := '0';
signal f_MEM_READY1    : std_logic := '0';
167 signal f_MEM_READY2  : std_logic := '0';
signal f_RESET0        : std_logic := '0';
169 signal f_RESET1      : std_logic := '0';
signal f_RESET2        : std_logic := '0';
171 signal f_NEXT_INSTR  : std_logic := '0';
signal f_TMR_ERROR     : std_logic := '0';
173 signal f_Err_Override : std_logic := '0';

175 begin
o_MEM_READ <= f_MEM_READ;
177 o_MEM_WRITE <= f_MEM_WRITE;
o_MEM_IN <= f_MEM_IN;

```

```

179 o_MEM_ADDRESS <= f_MEM_ADDRESS;
o_MEM_OUT0 <= f_MEM_OUT0;
181 o_MEM_OUT1 <= f_MEM_OUT1;
o_MEM_OUT2 <= f_MEM_OUT2;
183 o_MEM_READY0 <= f_MEM_READY0;
o_MEM_READY1 <= f_MEM_READY1;
185 o_MEM_READY2 <= f_MEM_READY2;
o_RESET0 <= f_RESET0;
187 o_RESET1 <= f_RESET1;
o_RESET2 <= f_RESET2;
189 o_NEXT_INSTR <= f_NEXT_INSTR;
o_TMR_ERROR <= f_TMR_ERROR;
191 o_Err_Override <= f_Err_Override;

— Compute the branch distance for correct processors when
  a type 0 error occurs
w_correct_branch_distance32 <= std_logic_vector(unsigned(
f_err_address) - unsigned(f_current_address));
195 — Branch instructions use 16 bit offsets that are shifted
  left by 2 to calculate the offset in MIPS
w_correct_branch_distance16 <= w_correct_branch_distance32
(17 downto 2);

197 — Compute the branch distance for the incorrect processor
  when a type 0 error occurs
w_incorrect_branch_distance32 <= std_logic_vector(unsigned(
199 f_err_address) - unsigned(k_124_32));
— Branch instructions use 16 bit offsets that are shifted
  left by 2 to calculate the offset in MIPS
201 w_incorrect_branch_distance16 <=
w_incorrect_branch_distance32(17 downto 2);

203 — Compute the branch distance for type 1 errors
w_err1_branch_distance32 <= std_logic_vector(unsigned(
f_err1_pc) - unsigned(k_124_32));
205 w_err1_branch_distance16 <= w_err1_branch_distance32(17
downto 2);

207 w_save_branch_distance32 <= std_logic_vector(unsigned(
f_save_address) - unsigned(f_save_current_address));
w_save_branch_distance16 <= w_save_branch_distance32(17
downto 2);
209

```

```

211 v fsm: process(i_clk , i_reset , f_v fsm_state ,
        i_MEM_READ0, i_MEM_READ1, i_MEM_READ2,
213 i_MEM_WRITE0, i_MEM_WRITE1, i_MEM_WRITE2,
        i_MEM_ADDRESS0, i_MEM_ADDRESS1,
i_MEM_ADDRESS2,
        i_MEM_IN0, i_MEM_IN1, i_MEM_IN2,
215 i_MEM_READY, i_MEM_OUT,
        f_done_wrong0 , f_done_wrong1 , f_done_wrong2 ,
217 f_done_all_wrong , f_done_save)

begin
219   if (i_reset = '1') then
        f_v fsm_state      <= s_v fsm_0;
        f_instr_count      <= (others => '0');
        f_wrong0           <= '0';
223 f_wrong1               <= '0';
        f_wrong2           <= '0';
        f_fsm_timer        <= (others => '0');
225   elsif rising_edge(i_clk) then
        case f_v fsm_state is
227         -- Determine when all three processors are
attempting to read or write.
        when s_v fsm_0 =>
229             if ((i_MEM_READ0 = '1') and (i_MEM_READ1 =
'1')) or ((i_MEM_READ1 = '1') and (i_MEM_READ2 = '1')) or
((i_MEM_READ0 = '1') and (i_MEM_READ2 = '1')) then
231                 f_v fsm_state <= s_v fsm_1;
                f_fsm_timer <= (others => '0');
                f_instr_count <= f_instr_count;
233             elsif f_fsm_timer > k_timeout then
                f_v fsm_state <= s_v fsm_err1;
                f_instr_count <= (others => '0');
235             else
                f_v fsm_state <= s_v fsm_0;
                f_fsm_timer <= f_fsm_timer + 1;
237                 f_instr_count <= f_instr_count;
                end if;
239
241         -- Determine whether an error has occurred with
one or more processors when reading an instruction from
memory
        when s_v fsm_1 =>
243             -- All three processors attempting to read a
word from the same memory address

```

```

247         if (iMEM_READ0 = '1') and (iMEM_READ1 =
'1') and (iMEM_READ2 = '1') and (iMEM_ADDRESS0 =
iMEM_ADDRESS1) and (iMEM_ADDRESS1 = iMEM_ADDRESS2) then
249             if f_instr_count >= k_save_point then
f_vfsm_state <= s_vfsm_save;
f_save_address <= iMEM_ADDRESS0;
f_instr_count <= (others => '0');
251             else
f_vfsm_state <= s_vfsm_2;
f_instr_count <= f_instr_count + 1;
f_err_address <= iMEM_ADDRESS0;
253             end if;
-- Processors 0 and 1 attempting to read a
word from the same memory address, and processor 2 in
error
257             elsif (iMEM_READ0 = '1') and (iMEM_READ1 =
'1') and (iMEM_ADDRESS0 = iMEM_ADDRESS1) then
f_vfsm_state <= s_vfsm_err0;
259             f_wrong2 <= '1';
f_err_address <= iMEM_ADDRESS0;
f_instr_count <= f_instr_count;
261             -- Processors 1 and 2 attempting to read a
word from the same memory address, and processor 0 in
error
263             elsif (iMEM_READ1 = '1') and (iMEM_READ2 =
'1') and (iMEM_ADDRESS1 = iMEM_ADDRESS2) then
f_vfsm_state <= s_vfsm_err0;
265             f_wrong0 <= '1';
f_err_address <= iMEM_ADDRESS1;
f_instr_count <= f_instr_count;
267             -- Processors 0 and 2 attempting to read a
word from the same memory address, and processor 1 in
error
269             elsif (iMEM_READ0 = '1') and (iMEM_READ2 =
'1') and (iMEM_ADDRESS0 = iMEM_ADDRESS2) then
f_vfsm_state <= s_vfsm_err0;
271             f_wrong1 <= '1';
f_err_address <= iMEM_ADDRESS0;
f_instr_count <= f_instr_count;
273             else
f_vfsm_state <= s_vfsm_err1;
f_instr_count <= (others => '0');
275             end if;
277

```

```

279      — Request data from memory by sending address
and awaiting response from memory.
    when s_vfsm_2 =>
281        f_instr_count <= f_instr_count;
        — Data has been returned from memory
283        if iMEM_READY = '1' then
            f_vfsm_state <= s_vfsm_3;
285            f_data <= iMEM_OUT;
            — Data has not yet returned from memory
287        else
            f_vfsm_state <= s_vfsm_2;
289        end if;

291      — Pass the instruction to all three processors
    when s_vfsm_3 =>
293        — If all three processors have received the
instruction ...
        if (iMEM_READ0 = '0') and (iMEM_READ1 = '0')
and (iMEM_READ2 = '0') then
295            f_instr_count <= f_instr_count;
            f_vfsm_state <= s_vfsm_3a;
297            f_fsm_timer <= (others => '0');
            — One of the processors has not received the
instruction prior to the timeout
299            elsif f_fsm_timer > k_timeout then
                f_vfsm_state <= s_vfsm_err1;
301                f_fsm_timer <= (others => '0');
                f_instr_count <= (others => '0');
303                — At least one of the three processors has
not received the instruction yet.
            else
305                f_vfsm_state <= s_vfsm_3;
                f_fsm_timer <= f_fsm_timer + 1;
307                f_instr_count <= f_instr_count;
            end if;
309      — Wait for memory ready signal to return to 0
    when s_vfsm_3a =>
311        if iMEM_READY = '0' then
            — Instruction is a load word instruction
313            if f_data(31 downto 26) = "10011" then —
iMEM_OUT(31 downto 26) = "100011" then
                f_vfsm_state <= s_vfsm_4;

```

```

315         — Instruction is a store word instruction
           elsif f_data(31 downto 26) = "101011" then
—iMEM_OUT(31 downto 26) = "101011" then
317             f_vfsm_state <= s_vfsm_8;
           — Instruction is not a load or store word
instruction
319       else
           f_vfsm_state <= s_vfsm_0;
321       end if;
           f_fsm_timer <= (others => '0');
323     else
           f_vfsm_state <= s_vfsm_3a;
325     end if;

327     — This is a load word operation. Wait for all
three processors to request data from memory
       when s_vfsm_4 =>
329         if ((iMEM_READ0 = '1') and (iMEM_READ1 =
'1')) or ((iMEM_READ1 = '1') and (iMEM_READ2 = '1')) or
((iMEM_READ0 = '1') and (iMEM_READ2 = '1')) then
           f_vfsm_state <= s_vfsm_5;
331           f_fsm_timer <= (others => '0');
           f_instr_count <= f_instr_count;
333         elsif f_fsm_timer > k_timeout then
           f_instr_count <= f_instr_count;
335           f_vfsm_state <= s_vfsm_err1;
           f_fsm_timer <= (others => '0');
337           f_instr_count <= (others => '0');
339         else
           f_vfsm_state <= s_vfsm_4;
           f_fsm_timer <= f_fsm_timer + 1;
341           f_instr_count <= f_instr_count;
           end if;
343

           — Check to make sure all three processors agree.
           If not, at least one processor is in error.
345       when s_vfsm_5 =>
           — All three processors attempting to read a
word from the same memory address
347         if (iMEM_READ0 = '1') and (iMEM_READ1 =
'1') and (iMEM_READ2 = '1') and (iMEM_ADDRESS0 =
iMEM_ADDRESS1) and (iMEM_ADDRESS1 = iMEM_ADDRESS2) then
           f_vfsm_state <= s_vfsm_6;

```

```

349         f_instr_count <= f_instr_count;
        — Processors 0 and 1 attempting to read a
word from the same memory address, and processor 2 in
error
351         elsif (iMEM_READ0 = '1') and (iMEM_READ1 =
'1') and (iMEM_ADDRESS0 = iMEM_ADDRESS1) then
            f_vfsm_state <= s_vfsm_5a;
353             f_wrong2 <= '1';
            f_instr_count <= f_instr_count;
355             — Processors 1 and 2 attempting to read a
word from the same memory address, and processor 0 in
error
            elsif (iMEM_READ1 = '1') and (iMEM_READ2 =
'1') and (iMEM_ADDRESS1 = iMEM_ADDRESS2) then
357                 f_vfsm_state <= s_vfsm_5a;
                f_wrong0 <= '1';
359                 f_instr_count <= f_instr_count;
                — Processors 0 and 2 attempting to read a
word from the same memory address, and processor 1 in
error
361                 elsif (iMEM_READ0 = '1') and (iMEM_READ2 =
'1') and (iMEM_ADDRESS0 = iMEM_ADDRESS2) then
                    f_vfsm_state <= s_vfsm_5a;
363                     f_wrong1 <= '1';
                    f_instr_count <= f_instr_count;
365                 else
                    f_vfsm_state <= s_vfsm_err1;
367                     f_instr_count <= (others => '0');
                end if;
369
            — When a read error has occurred because one
processor disagrees, send the mem ready signal and wait
for
371            — the mem read signals to return to zero
            when s_vfsm_5a =>
373                if (f_wrong0 = '1') then
                    if (iMEM_READ1 = '0') and (iMEM_READ2 =
'0') then
375                        f_vfsm_state <= s_vfsm_err0;
                        f_fsm_timer <= (others => '0');
377                        f_instr_count <= f_instr_count;
                    elsif f_fsm_timer > k_timeout then
                        f_vfsm_state <= s_vfsm_err1;
379

```

```

381         f_fsm_timer <= (others => '0');
382         f_instr_count <= (others => '0');
383     else
384         f_vfsm_state <= s_vfsm_5a;
385         f_fsm_timer <= f_fsm_timer + 1;
386         f_instr_count <= f_instr_count;
387     end if;
388 elsif (f_wrong1 = '1') then
389     if (i_MEM_READ0 = '0') and (i_MEM_READ2 =
'0') then
390         f_vfsm_state <= s_vfsm_err0;
391         f_fsm_timer <= (others => '0');
392         f_instr_count <= f_instr_count;
393     elsif f_fsm_timer > k_timeout then
394         f_vfsm_state <= s_vfsm_err1;
395         f_fsm_timer <= (others => '0');
396         f_instr_count <= (others => '0');
397     else
398         f_vfsm_state <= s_vfsm_5a;
399         f_fsm_timer <= f_fsm_timer + 1;
400         f_instr_count <= f_instr_count;
401     end if;
402 elsif (f_wrong2 = '1') then
403     if (i_MEM_READ0 = '0') and (i_MEM_READ1 =
'0') then
404         f_vfsm_state <= s_vfsm_err0;
405         f_fsm_timer <= (others => '0');
406         f_instr_count <= f_instr_count;
407     elsif f_fsm_timer > k_timeout then
408         f_vfsm_state <= s_vfsm_err1;
409         f_fsm_timer <= (others => '0');
410         f_instr_count <= (others => '0');
411     else
412         f_vfsm_state <= s_vfsm_5a;
413         f_fsm_timer <= f_fsm_timer + 1;
414         f_instr_count <= f_instr_count;
415     end if;
416 end if;
417
418     — Request data from memory by sending address
419     and awaiting response from memory.
420     when s_vfsm_6 =>

```

```

421         f_instr_count <= f_instr_count;
422         — Data has been returned from memory
423         if (iMEM_READY = '1') then
424             f_vfsm_state <= s_vfsm_7;
425             f_data <= iMEM_OUT;
426         — Data has not yet returned from memory
427         else
428             f_vfsm_state <= s_vfsm_6;
429         end if;
430
431         — Pass data back to all three processors
432         when s_vfsm_7 =>
433             — If all three processors have received the
data...
434             if (iMEM_READ0 = '0') and (iMEM_READ1 = '0')
and (iMEM_READ2 = '0') then
435                 f_vfsm_state <= s_vfsm_7a;
436                 f_instr_count <= f_instr_count;
437             elsif f_fsm_timer > k_timeout then
438                 f_vfsm_state <= s_vfsm_err1;
439                 f_fsm_timer <= (others => '0');
440                 f_instr_count <= (others => '0');
441             — At least one of the three processors has
not received the instruction.
442             else
443                 f_vfsm_state <= s_vfsm_7;
444                 f_fsm_timer <= f_fsm_timer + 1;
445                 f_instr_count <= f_instr_count;
446             end if;
447
448         when s_vfsm_7a =>
449             if (iMEM_READY = '0') then
450                 f_vfsm_state <= s_vfsm_0;
451             else
452                 f_vfsm_state <= s_vfsm_7a;
453             end if;
454             f_instr_count <= f_instr_count;
455             f_fsm_timer <= (others => '0');
456
457         — This is a store word operation. Wait for at
least two of three processors to send data to memory
458         when s_vfsm_8 =>
459             if ((iMEM_WRITE0 = '1') and (iMEM_WRITE1 =

```

```

459      '1')) or ((iMEM_WRITE1 = '1') and (iMEM_WRITE2 = '1'))
or ((iMEM_WRITE0 = '1') and (iMEM_WRITE2 = '1')) then
461          f_vfsm_state <= s_vfsm_9;
          f_fsm_timer <= (others => '0');
          f_instr_count <= f_instr_count;
463      elsif f_fsm_timer > k_timeout then
          f_vfsm_state <= s_vfsm_err1;
          f_fsm_timer <= (others => '0');
465          f_instr_count <= (others => '0');
      else
467          f_vfsm_state <= s_vfsm_8;
          f_fsm_timer <= f_fsm_timer + 1;
469          f_instr_count <= f_instr_count;
      end if;

471      — Check to make sure all three processors agree.
      If not, at least one processor is in error.
473      when s_vfsm_9 =>
          — All three processors attempting to write a
          word to the same memory address
          if (iMEM_WRITE0 = '1') and (iMEM_WRITE1 =
475      '1') and (iMEM_WRITE2 = '1') and (iMEM_ADDRESS0 =
iMEM_ADDRESS1) and (iMEM_ADDRESS1 = iMEM_ADDRESS2) then
          — Check to make sure all three processors
          are attempting to write the same word to memory
477          if (iMEM_IN0 = iMEM_IN1) and (iMEM_IN1 =
iMEM_IN2) then
              f_vfsm_state <= s_vfsm_10;
479              f_instr_count <= f_instr_count;
          — Processors 0 and 1 attempting to write
          the same data and processor 2 attempting to write
          different data
481          elsif (iMEM_IN0 = iMEM_IN1) then
              f_vfsm_state <= s_vfsm_9a;
483              f_wrong2 <= '1';
              f_instr_count <= f_instr_count;
          — Processors 1 and 2 attempting to write
485      the same data and processor 0 attempting to write
          different data
          elsif (iMEM_IN1 = iMEM_IN2) then
487              f_vfsm_state <= s_vfsm_9a;
              f_wrong0 <= '1';
489              f_instr_count <= f_instr_count;

```

```

491         — Processors 0 and 2 attempting to write
the same data and processor 1 attempting to write
different data
492         elsif (i_MEM_IN0 = i_MEM_IN2) then
493             f_vfsm_state <= s_vfsm_9a;
494             f_wrong1 <= '1';
495             f_instr_count <= f_instr_count;
— All processors are attempting to write
different data
496         else
497             f_vfsm_state <= s_vfsm_err1;
498             f_instr_count <= (others => '0');
499         end if;
— Processors 0 and 1 attempting to write a
word to the same memory address, and processor 2 in error
500         elsif (i_MEM_WRITE0 = '1') and (i_MEM_WRITE1
= '1') and (i_MEM_ADDRESS0 = i_MEM_ADDRESS1) then
— Check to make sure processors 0 and 1
are attempting to write the same data
501         if (i_MEM_IN0 = i_MEM_IN1) then
502             f_vfsm_state <= s_vfsm_9a;
503             f_wrong2 <= '1';
504             f_err_address <= i_MEM_ADDRESS0;
505             f_instr_count <= f_instr_count;
506         else
507             f_vfsm_state <= s_vfsm_err1;
508             f_instr_count <= (others => '0');
509         end if;
— Processors 1 and 2 attempting to write a
word to the same memory address, and processor 0 in error
510         elsif (i_MEM_WRITE1 = '1') and (i_MEM_WRITE2
= '1') and (i_MEM_ADDRESS1 = i_MEM_ADDRESS2) then
— Check to make sure processors 1 and 2
are attempting to write the same data
511         if (i_MEM_IN1 = i_MEM_IN2) then
512             f_vfsm_state <= s_vfsm_9a;
513             f_wrong0 <= '1';
514             f_err_address <= i_MEM_ADDRESS1;
515             f_instr_count <= f_instr_count;
516         else
517             f_vfsm_state <= s_vfsm_err1;
518             f_instr_count <= (others => '0');
519         end if;
520
521
522
523

```

```

525      — Processors 0 and 2 attempting to read a
word from the same memory address, and processor 1 in
error
      elsif (iMEM_WRITE0 = '1') and (iMEM_WRITE2
= '1') and (iMEM_ADDRESS0 = iMEM_ADDRESS2) then
      — Check to make sure processors 0 and 2
are attempting to write the same data
527      if (iMEM_IN0 = iMEM_IN2) then
          f_vfsm_state <= s_vfsm_9a;
529          f_wrong1 <= '1';
          f_err_address <= iMEM_ADDRESS0;
          f_instr_count <= f_instr_count;
531      else
          f_vfsm_state <= s_vfsm_err1;
533          f_instr_count <= (others => '0');
535      end if;
      else
537          f_vfsm_state <= s_vfsm_err1;
          f_instr_count <= (others => '0');
539      end if;

541      — When a write error has occurred because one
processor disagrees, send the mem ready signal and wait
for
      — the mem write signals to return to zero
543      when s_vfsm_9a =>
          if (f_wrong0 = '1') then
545              if (iMEM_WRITE1 = '0') and (iMEM_WRITE2 =
'0') then
                  f_vfsm_state <= s_vfsm_err0;
547                  f_fsm_timer <= (others => '0');
                  f_instr_count <= f_instr_count;
549              elsif f_fsm_timer > k_timeout then
                  f_vfsm_state <= s_vfsm_err1;
551                  f_fsm_timer <= (others => '0');
                  f_instr_count <= (others => '0');
553              else
                  f_vfsm_state <= s_vfsm_9a;
555                  f_fsm_timer <= f_fsm_timer + 1;
                  f_instr_count <= f_instr_count;
557              end if;
          elsif (f_wrong1 = '1') then
559              if (iMEM_WRITE0 = '0') and (iMEM_WRITE2 =

```

```

561      '0') then
          f_vfsm_state <= s_vfsm_err0;
          f_fsm_timer <= (others => '0');
          f_instr_count <= f_instr_count;
563      elsif f_fsm_timer > k_timeout then
          f_vfsm_state <= s_vfsm_err1;
565          f_fsm_timer <= (others => '0');
          f_instr_count <= (others => '0');
567      else
          f_vfsm_state <= s_vfsm_9a;
          f_fsm_timer <= f_fsm_timer + 1;
569          f_instr_count <= f_instr_count;
571      end if;

573      elsif (f_wrong2 = '1') then
          if (iMEM_WRITE0 = '0') and (iMEM_WRITE1 =
575      '0') then
              f_vfsm_state <= s_vfsm_err0;
              f_fsm_timer <= (others => '0');
              f_instr_count <= f_instr_count;
577          elsif f_fsm_timer > k_timeout then
              f_vfsm_state <= s_vfsm_err1;
              f_fsm_timer <= (others => '0');
579              f_instr_count <= (others => '0');
581          else
              f_vfsm_state <= s_vfsm_9a;
              f_fsm_timer <= f_fsm_timer + 1;
583              f_instr_count <= f_instr_count;
585          end if;
587      end if;
      — Write data to memory by sending address and
data and awaiting response from memory.
589      when s_vfsm_10 =>
          f_instr_count <= f_instr_count;
          — Data has been written to memory
          if (iMEM_READY = '1') then
591              f_vfsm_state <= s_vfsm_11;
          — Data has not yet been written to memory
593          else
              f_vfsm_state <= s_vfsm_10;
595          end if;
597
599      — Pass MEMREADY signal back to all processors

```

```

601         when s_vfsm_11 =>
            — If all three processors have received the
data...
            if (iMEM_WRITE0 = '0') and (iMEM_WRITE1 =
'0') and (iMEM_WRITE2 = '0') then
603                 f_vfsm_state <= s_vfsm_12;
                 f_fsm_timer <= (others => '0');
605                 f_instr_count <= f_instr_count;
            elsif f_fsm_timer > k_timeout then
607                 f_vfsm_state <= s_vfsm_err1;
                 f_fsm_timer <= (others => '0');
609                 f_instr_count <= (others => '0');
            — At least one of the three processors has
not received the instruction.
611            else
                 f_vfsm_state <= s_vfsm_11;
613                 f_fsm_timer <= f_fsm_timer + 1;
                 f_instr_count <= f_instr_count;
615            end if;

617         when s_vfsm_12 =>
            if (iMEM_READY = '0') then
619                 f_vfsm_state <= s_vfsm_0;
            else
621                 f_vfsm_state <= s_vfsm_12;
            end if;
623                 f_fsm_timer <= (others => '0');
                 f_instr_count <= f_instr_count;
625

            — Enter this state when one of three processors
are wrong
627         when s_vfsm_err0 =>
            if (f_wrong0 = '1') and (f_done_wrong0 = '1')
then
629                 if (iMEM_READ0 = '1') then
                     f_vfsm_state <= s_vfsm_0;
631                     f_wrong0 <= '0';
                     f_instr_count <= f_instr_count;
633                 elsif f_fsm_timer > k_timeout then
                     f_vfsm_state <= s_vfsm_err1;
635                     f_fsm_timer <= (others => '0');
                     f_instr_count <= (others => '0');
637                 else

```

```

639         f_vfsm_state <= s_vfsm_err0;
        f_fsm_timer <= f_fsm_timer + 1;
        f_instr_count <= f_instr_count;
641     end if;
    elsif (f_wrong1 = '1') and (f_done_wrong1 =
'1') then
643         if (i_MEM_READ1 = '1') then
            f_vfsm_state <= s_vfsm_0;
645             f_wrong1 <= '0';
            f_instr_count <= f_instr_count;
647         elsif f_fsm_timer > k_timeout then
            f_vfsm_state <= s_vfsm_err1;
649             f_fsm_timer <= (others => '0');
            f_instr_count <= (others => '0');
651         else
            f_vfsm_state <= s_vfsm_err0;
653             f_fsm_timer <= f_fsm_timer + 1;
            f_instr_count <= f_instr_count;
655         end if;
    elsif (f_wrong2 = '1') and (f_done_wrong2 =
'1') then
657         if (i_MEM_READ2 = '1') then
            f_vfsm_state <= s_vfsm_0;
659             f_wrong2 <= '0';
            f_instr_count <= f_instr_count;
661         elsif f_fsm_timer > k_timeout then
            f_vfsm_state <= s_vfsm_err1;
663             f_fsm_timer <= (others => '0');
            f_instr_count <= (others => '0');
665         else
            f_vfsm_state <= s_vfsm_err0;
667             f_fsm_timer <= f_fsm_timer + 1;
            f_instr_count <= f_instr_count;
669         end if;
    elsif f_verr0_state = s_verr0_err1 then
671         f_vfsm_state <= s_vfsm_err1;
        f_wrong0 <= '0';
673         f_wrong1 <= '0';
        f_wrong2 <= '0';
675         f_fsm_timer <= (others => '0');
        f_instr_count <= (others => '0');
677     else
        f_vfsm_state <= s_vfsm_err0;

```

```

679         f_instr_count <= f_instr_count;
        end if;
681
        — Enter this state when all three processors are
wrong
683         when s_vfsm_err1 =>
            f_instr_count <= (others => '0');
685             if (f_done_all_wrong = '1') then
                f_vfsm_state <= s_vfsm_0;
687             else
                f_vfsm_state <= s_vfsm_err1;
689             end if;
        — Enter this state when it is time to create a
save point from which a recovery may take place
691         when s_vfsm_save =>
            f_instr_count <= (others => '0');
693             if (f_done_save = '1') then
                f_vfsm_state <= s_vfsm_0;
695             elsif (f_vsave_state = s_vsave_err1) then
                f_vfsm_state <= s_vfsm_err1;
697             else
                f_vfsm_state <= s_vfsm_save;
699             end if;
        — This should never happen
701         when others =>

703
        end case;
705     end if;
end process vfsm;

707
err0: process(i_clk , i_reset , f_vfsm_state , f_verr0_state ,
709             i_MEM_READ0, i_MEM_READ1, i_MEM_READ2,
                i_MEM_WRITE0, i_MEM_WRITE1, i_MEM_WRITE2,
711             i_MEM_ADDRESS0, i_MEM_ADDRESS1,
i_MEM_ADDRESS2,
                i_MEM_IN0, i_MEM_IN1, i_MEM_IN2,
713             i_MEM_READY, i_MEM_OUT,
                f_err0_timer)
715
begin
    if (i_reset = '1') then
717         f_verr0_state <= s_verr0_0;
        f_done_wrong0 <= '0';

```

```

719         f_done_wrong1 <= '0';
720         f_done_wrong2 <= '0';
721         f_err0_timer <= (others => '0');
722         f_err0_count <= (others => '0');
723     elsif rising_edge(i_clk) then
724         if f_vfsm_state = s_vfsm_err0 then
725             case f_verr0_state is
726                 -- Waiting for an error to occur in one of the
727                 processors
728                 when s_verr0_0 =>
729                     f_verr0_state <= s_verr0_1;
730
731                 -- Reset the processor that is in error
732                 when s_verr0_1 =>
733                     f_verr0_state <= s_verr0_2;
734
735                 --| Copy data from correct processors to
736                 incorrect processor one register at a time
737                 -- Wait for correct processors to attempt to
738                 retrieve an instruction from memory'
739                 when s_verr0_2 =>
740                     -- If processor 0 is wrong
741                     if (f_wrong0 = '1') then
742                         if (iMEM_READ1 = '1') and (iMEM_READ2
743 = '1') and (iMEM_ADDRESS1 = iMEM_ADDRESS2) then
744                             f_verr0_state <= s_verr0_3;
745                             f_err0_timer <= (others => '0');
746                             f_current_address <= iMEM_ADDRESS1;
747                         elsif (iMEM_READ1 = '1') and (
748 iMEM_READ2 = '1') then
749                             f_verr0_state <= s_verr0_err1;
750                             f_err0_timer <= (others => '0');
751                         elsif (f_err0_timer > k_timeout) then
752                             f_verr0_state <= s_verr0_err1;
753                             f_err0_timer <= (others => '0');
754                         else
755                             f_verr0_state <= s_verr0_2;
756                             f_err0_timer <= f_err0_timer + 1;
757                         end if;
758                     -- If processor 1 is wrong
759                     elsif (f_wrong1 = '1') then
760                         if (iMEM_READ0 = '1') and (iMEM_READ2
761 = '1') and (iMEM_ADDRESS0 = iMEM_ADDRESS2) then

```

```

757         f_verr0_state <= s_verr0_3;
758         f_err0_timer <= (others => '0');
759         f_current_address <= iMEM_ADDRESS0;
760     elsif (iMEM_READ0 = '1') and (
iMEM_READ2 = '1') then
761         f_verr0_state <= s_verr0_err1;
762         f_err0_timer <= (others => '0');
763     elsif (f_err0_timer > k_timeout) then
764         f_verr0_state <= s_verr0_err1;
765         f_err0_timer <= (others => '0');
766     else
767         f_verr0_state <= s_verr0_2;
768         f_err0_timer <= f_err0_timer + 1;
769     end if;
770     -- If processor 2 is wrong
771     else
772         if (iMEM_READ0 = '1') and (iMEM_READ1
= '1') and (iMEM_ADDRESS0 = iMEM_ADDRESS1) then
773             f_verr0_state <= s_verr0_3;
774             f_err0_timer <= (others => '0');
775             f_current_address <= iMEM_ADDRESS0;
776         elsif (iMEM_READ0 = '1') and (
iMEM_READ1 = '1') then
777             f_verr0_state <= s_verr0_err1;
778             f_err0_timer <= (others => '0');
779         elsif (f_err0_timer > k_timeout) then
780             f_verr0_state <= s_verr0_err1;
781             f_err0_timer <= (others => '0');
782         else
783             f_verr0_state <= s_verr0_2;
784             f_err0_timer <= f_err0_timer + 1;
785         end if;
786     end if;
787     -- Issue store word commands to the correct
processors
788     when s_verr0_3 =>
789         -- If processor 0 is wrong
790         if (f_wrong0 = '1') then
791             if (iMEM_READ1 = '0') and (iMEM_READ2
= '0') then
792                 f_verr0_state <= s_verr0_4;
793                 f_err0_timer <= (others => '0');

```

```

795         elsif (f_err0_timer > k_timeout) then
797             f_verr0_state <= s_verr0_err1;
799             f_err0_timer <= (others => '0');
801         else
803             f_verr0_state <= s_verr0_3;
805             f_err0_timer <= f_err0_timer + 1;
807         end if;
809 — If processor 1 is wrong
811 elsif (f_wrong1 = '1') then
813     if (iMEM_READ0 = '0') and (iMEM_READ2
= '0') then
815         f_verr0_state <= s_verr0_4;
817         f_err0_timer <= (others => '0');
819         elsif (f_err0_timer > k_timeout) then
821             f_verr0_state <= s_verr0_err1;
823             f_err0_timer <= (others => '0');
825         else
827             f_verr0_state <= s_verr0_3;
829             f_err0_timer <= f_err0_timer + 1;
831         end if;
833 — If processor 2 is wrong
835 else
837     if (iMEM_READ0 = '0') and (iMEM_READ1
= '0') then
839         f_verr0_state <= s_verr0_4;
841         f_err0_timer <= (others => '0');
843         elsif (f_err0_timer > k_timeout) then
845             f_verr0_state <= s_verr0_err1;
847             f_err0_timer <= (others => '0');
849         else
851             f_verr0_state <= s_verr0_3;
853             f_err0_timer <= f_err0_timer + 1;
855         end if;
857     end if;
859 — Wait for correct processors to issue store
word requests
861 when s_verr0_4 =>
863     — If processor 0 is wrong
865     if (f_wrong0 = '1') then
867         if (iMEM_WRITE1 = '1') and (
iMEM_WRITE2 = '1') then
869             f_verr0_state <= s_verr0_5;

```

```

833         f_err0_timer <= (others => '0');
      elsif (f_err0_timer > k_timeout) then
835         f_verr0_state <= s_verr0_err1;
         f_err0_timer <= (others => '0');
837     else
         f_verr0_state <= s_verr0_4;
839         f_err0_timer <= f_err0_timer + 1;
      end if;
841   -- If processor 1 is wrong
      elsif (f_wrong1 = '1') then
843         if (iMEM_WRITE0 = '1') and (
iMEM_WRITE2 = '1') then
            f_verr0_state <= s_verr0_5;
845            f_err0_timer <= (others => '0');
            elsif (f_err0_timer > k_timeout) then
847                f_verr0_state <= s_verr0_err1;
                f_err0_timer <= (others => '0');
849            else
                f_verr0_state <= s_verr0_4;
851                f_err0_timer <= f_err0_timer + 1;
            end if;
853   -- If processor 2 is wrong
      else
855         if (iMEM_WRITE0 = '1') and (
iMEM_WRITE1 = '1') then
            f_verr0_state <= s_verr0_5;
857            f_err0_timer <= (others => '0');
            elsif (f_err0_timer > k_timeout) then
859                f_verr0_state <= s_verr0_err1;
                f_err0_timer <= (others => '0');
861            else
                f_verr0_state <= s_verr0_4;
863                f_err0_timer <= f_err0_timer + 1;
            end if;
865        end if;

867   -- Check that data from the correct processors
   is a match and wait for write signals to return to 0
      when s_verr0_5 =>
869       -- If processor 0 is wrong
         if (f_wrong0 = '1') then
871             if (iMEM_WRITE1 = '0') and (
iMEM_WRITE2 = '0') then

```

```

873         if (i_MEM_IN1 = i_MEM_IN2) then
875             f_verr0_state <= s_verr0_6;
877             f_err0_data <= i_MEM_IN1;
879             f_err0_timer <= (others => '0');
881         else
883             f_verr0_state <= s_verr0_err1;
885             f_err0_timer <= (others => '0');
887         end if;
889     elsif (f_err0_timer > k_timeout) then
891         f_verr0_state <= s_verr0_err1;
893         f_err0_timer <= (others => '0');
895     else
897         f_verr0_state <= s_verr0_5;
899         f_err0_timer <= f_err0_timer + 1;
901     end if;
903     -- If processor 1 is wrong
905     elsif (f_wrong1 = '1') then
907         if (i_MEM_WRITE0 = '0') and (
i_MEM_WRITE2 = '0') then
909             if (i_MEM_IN0 = i_MEM_IN2) then
911                 f_verr0_state <= s_verr0_6;
913                 f_err0_data <= i_MEM_IN0;
915             else
917                 f_verr0_state <= s_verr0_err1;
919                 f_err0_timer <= (others => '0');
921             end if;
923         elsif (f_err0_timer > k_timeout) then
925             f_verr0_state <= s_verr0_err1;
927             f_err0_timer <= (others => '0');
929         else
931             f_verr0_state <= s_verr0_5;
933             f_err0_timer <= f_err0_timer + 1;
935         end if;
937     -- If processor 2 is wrong
939     else
941         if (i_MEM_WRITE0 = '0') and (
i_MEM_WRITE1 = '0') then
943             if (i_MEM_IN0 = i_MEM_IN1) then
945                 f_verr0_state <= s_verr0_6;
947                 f_err0_data <= i_MEM_IN0;
949             else
951                 f_verr0_state <= s_verr0_err1;
953                 f_err0_timer <= (others => '0');
955             end if;
957         else
959             f_verr0_state <= s_verr0_5;
961             f_err0_timer <= f_err0_timer + 1;
963         end if;
965     end if;
967 end if;
969
971
973
975
977
979
981
983
985
987
989
991

```

```

913         end if;
          elsif (f_err0_timer > k_timeout) then
915             f_verr0_state <= s_verr0_err1;
             f_err0_timer <= (others => '0');
917         else
             f_verr0_state <= s_verr0_5;
919             f_err0_timer <= f_err0_timer + 1;
             end if;
921         end if;

923         -- Wait for incorrect processor to attempt to
retrieve an instruction from memory
         when s_verr0_6 =>
925             -- If processor 0 is wrong
             if (f_wrong0 = '1') then
927                 if (iMEM_READ0 = '1') then
                     f_verr0_state <= s_verr0_7;
929                     f_err0_timer <= (others => '0');
                     elsif (f_err0_timer > k_timeout) then
931                         f_verr0_state <= s_verr0_err1;
                         f_err0_timer <= (others => '0');
933                     else
                         f_verr0_state <= s_verr0_6;
935                         f_err0_timer <= f_err0_timer + 1;
                         end if;
937                 -- If processor 1 is wrong
                 elsif (f_wrong1 = '1') then
939                     if (iMEM_READ1 = '1') then
                         f_verr0_state <= s_verr0_7;
941                         f_err0_timer <= (others => '0');
                         elsif (f_err0_timer > k_timeout) then
943                             f_verr0_state <= s_verr0_err1;
                             f_err0_timer <= (others => '0');
945                         else
                             f_verr0_state <= s_verr0_6;
947                             f_err0_timer <= f_err0_timer + 1;
                             end if;
949                 -- If processor 2 is wrong
                 else
951                     if (iMEM_READ2 = '1') then
                         f_verr0_state <= s_verr0_7;
953                         f_err0_timer <= (others => '0');
                         elsif (f_err0_timer > k_timeout) then

```

```

955         f_verr0_state <= s_verr0_err1;
          f_err0_timer <= (others => '0');
957     else
          f_verr0_state <= s_verr0_6;
959         f_err0_timer <= f_err0_timer + 1;
          end if;
961     end if;

963     — Issue load word command to the incorrect
processor
    when s_verr0_7 =>
965         — If processor 0 is wrong
          if (f_wrong0 = '1') then
967             if (iMEM_READ0 = '0') then
                  f_verr0_state <= s_verr0_8;
969                 f_err0_timer <= (others => '0');
                elsif (f_err0_timer > k_timeout) then
971                     f_verr0_state <= s_verr0_err1;
                     f_err0_timer <= (others => '0');
973                 else
                     f_verr0_state <= s_verr0_7;
975                     f_err0_timer <= f_err0_timer + 1;
                     end if;
977             — If processor 1 is wrong
                elsif (f_wrong1 = '1') then
979                 if (iMEM_READ1 = '0') then
                     f_verr0_state <= s_verr0_8;
981                     f_err0_timer <= (others => '0');
                    elsif (f_err0_timer > k_timeout) then
983                         f_verr0_state <= s_verr0_err1;
                         f_err0_timer <= (others => '0');
985                     else
                         f_verr0_state <= s_verr0_7;
987                         f_err0_timer <= f_err0_timer + 1;
                         end if;
989                 — If processor 2 is wrong
                else
991                 if (iMEM_READ2 = '0') then
                     f_verr0_state <= s_verr0_8;
993                     f_err0_timer <= (others => '0');
                    elsif (f_err0_timer > k_timeout) then
995                         f_verr0_state <= s_verr0_err1;
                         f_err0_timer <= (others => '0');

```

```

997         else
999             f_verr0_state <= s_verr0_7;
1000             f_err0_timer <= f_err0_timer + 1;
1001         end if;
1002     end if;
1003
1004     — Wait for incorrect processor to attempt to
1005     load the requested data from memory
1006     when s_verr0_8 =>
1007         — If processor 0 is wrong
1008         if (f_wrong0 = '1') then
1009             if (iMEM_READ0 = '1') then
1010                 f_verr0_state <= s_verr0_9;
1011                 f_err0_timer <= (others => '0');
1012             elsif (f_err0_timer > k_timeout) then
1013                 f_verr0_state <= s_verr0_err1;
1014                 f_err0_timer <= (others => '0');
1015             else
1016                 f_verr0_state <= s_verr0_8;
1017                 f_err0_timer <= f_err0_timer + 1;
1018             end if;
1019         — If processor 1 is wrong
1020         elsif (f_wrong1 = '1') then
1021             if (iMEM_READ1 = '1') then
1022                 f_verr0_state <= s_verr0_9;
1023                 f_err0_timer <= (others => '0');
1024             elsif (f_err0_timer > k_timeout) then
1025                 f_verr0_state <= s_verr0_err1;
1026                 f_err0_timer <= (others => '0');
1027             else
1028                 f_verr0_state <= s_verr0_8;
1029                 f_err0_timer <= f_err0_timer + 1;
1030             end if;
1031         — If processor 2 is wrong
1032         else
1033             if (iMEM_READ2 = '1') then
1034                 f_verr0_state <= s_verr0_9;
1035                 f_err0_timer <= (others => '0');
1036             elsif (f_err0_timer > k_timeout) then
1037                 f_verr0_state <= s_verr0_err1;
1038                 f_err0_timer <= (others => '0');
1039             else
1040                 f_verr0_state <= s_verr0_8;

```

```

1039         f_err0_timer <= f_err0_timer + 1;
1041     end if;
1043     — Return desired data to the incorrect
processor
    when s_verr0_9 =>
1045         — If processor 0 is wrong
        if (f_wrong0 = '1') then
1047             if (iMEM_READ0 = '0') then
                — If all the registers have been
copied
1049                 if (to_integer(f_err0_count) = 30)
then
1051                     f_verr0_state <= s_verr0_10;
                     f_err0_count <= (others => '0');
                    — If all the registers have not been
copied
1053                     else
                        f_verr0_state <= s_verr0_2;
                        f_err0_count <= f_err0_count + 1;
                    end if;
1055                     f_err0_timer <= (others => '0');
                    elsif (f_err0_timer > k_timeout) then
1057                         f_verr0_state <= s_verr0_err1;
                        f_err0_timer <= (others => '0');
1059                     else
                        f_verr0_state <= s_verr0_9;
                        f_err0_timer <= f_err0_timer + 1;
1061                     end if;
                    — If processor 1 is wrong
                    elsif (f_wrong1 = '1') then
1063                         if (iMEM_READ1 = '0') then
                            — If all the registers have been
copied
1065                             if (to_integer(f_err0_count) = 30)
then
1067                                 f_verr0_state <= s_verr0_10;
                                    f_err0_count <= (others => '0');
                                — If all the registers have not been
copied
1069                                    else
                                        f_verr0_state <= s_verr0_2;
1071
1073

```

```

1075         f_err0_count <= f_err0_count + 1;
1076     end if;
1077     f_err0_timer <= (others => '0');
1078     elsif (f_err0_timer > k_timeout) then
1079         f_verr0_state <= s_verr0_err1;
1080         f_err0_timer <= (others => '0');
1081     else
1082         f_verr0_state <= s_verr0_9;
1083         f_err0_timer <= f_err0_timer + 1;
1084     end if;
1085     -- If processor 2 is wrong
1086     else
1087         if (iMEM_READ2 = '0') then
1088             -- If all the registers have been
1089             copied
1090             then
1091                 f_verr0_state <= s_verr0_10;
1092                 f_err0_count <= (others => '0');
1093                 -- If all the registers have not been
1094                 copied
1095                 else
1096                     f_verr0_state <= s_verr0_2;
1097                     f_err0_count <= f_err0_count + 1;
1098                 end if;
1099                 f_err0_timer <= (others => '0');
1100             elsif (f_err0_timer > k_timeout) then
1101                 f_verr0_state <= s_verr0_err1;
1102                 f_err0_timer <= (others => '0');
1103             else
1104                 f_verr0_state <= s_verr0_9;
1105                 f_err0_timer <= f_err0_timer + 1;
1106             end if;
1107         end if;
1108     end if;
1109     --| Return processor's program counters to the
1110     point at which the error was encountered
1111     --Wait for correct processors to request next
    instruction
    when s_verr0_10 =>
        -- If processor 0 is wrong
        if (f_wrong0 = '1') then
            if (iMEM_READ1 = '1') and (iMEM_READ2

```

```

1113 = '1') and (iMEM_ADDRESS1 = iMEM_ADDRESS2) then
        f_verr0_state <= s_verr0_11;
        f_err0_timer <= (others => '0');
1115        f_current_address <= iMEM_ADDRESS1;
        elsif (iMEM_READ1 = '1') and (
iMEM_READ2 = '1') then
1117            f_verr0_state <= s_verr0_err1;
            f_err0_timer <= (others => '0');
1119            elsif (f_err0_timer > k_timeout) then
                f_verr0_state <= s_verr0_err1;
1121                f_err0_timer <= (others => '0');
            else
1123                f_verr0_state <= s_verr0_10;
                f_err0_timer <= f_err0_timer + 1;
1125            end if;
            -- If processor 1 is wrong
1127            elsif (f_wrong1 = '1') then
                if (iMEM_READ0 = '1') and (iMEM_READ2
= '1') and (iMEM_ADDRESS0 = iMEM_ADDRESS2) then
1129                    f_verr0_state <= s_verr0_11;
                    f_err0_timer <= (others => '0');
1131                    f_current_address <= iMEM_ADDRESS0;
                    elsif (iMEM_READ0 = '1') and (
iMEM_READ2 = '1') then
1133                        f_verr0_state <= s_verr0_err1;
                        f_err0_timer <= (others => '0');
1135                        elsif (f_err0_timer > k_timeout) then
                            f_verr0_state <= s_verr0_err1;
1137                            f_err0_timer <= (others => '0');
                        else
1139                            f_verr0_state <= s_verr0_10;
                            f_err0_timer <= f_err0_timer + 1;
1141                        end if;
                        -- If processor 2 is wrong
1143                        else
                            if (iMEM_READ0 = '1') and (iMEM_READ1
= '1') and (iMEM_ADDRESS0 = iMEM_ADDRESS1) then
1145                                f_verr0_state <= s_verr0_11;
                                f_err0_timer <= (others => '0');
1147                                f_current_address <= iMEM_ADDRESS0;
                                elsif (iMEM_READ0 = '1') and (
iMEM_READ1 = '1') then
1149                                    f_verr0_state <= s_verr0_err1;

```

```

1151         f_err0_timer <= (others => '0');
1152     elsif (f_err0_timer > k_timeout) then
1153         f_verr0_state <= s_verr0_err1;
1154         f_err0_timer <= (others => '0');
1155     else
1156         f_verr0_state <= s_verr0_10;
1157         f_err0_timer <= f_err0_timer + 1;
1158     end if;
1159 end if;

1160     -- Issue branch command to correct processors
1161     -- to return to the instruction call
1162     -- that caused the error. Wait for processors
1163     -- to acknowledge command
1164     when s_verr0_11 =>
1165         -- If processor 0 is wrong
1166         if (f_wrong0 = '1') then
1167             if (iMEM_READ1 = '0') and (iMEM_READ2
= '0') then
1168                 f_verr0_state <= s_verr0_12;
1169                 f_err0_timer <= (others => '0');
1170             elsif (f_err0_timer > k_timeout) then
1171                 f_verr0_state <= s_verr0_err1;
1172                 f_err0_timer <= (others => '0');
1173             else
1174                 f_verr0_state <= s_verr0_11;
1175                 f_err0_timer <= f_err0_timer + 1;
1176             end if;
1177         -- If processor 1 is wrong
1178         elsif (f_wrong1 = '1') then
1179             if (iMEM_READ0 = '0') and (iMEM_READ2
= '0') then
1180                 f_verr0_state <= s_verr0_12;
1181                 f_err0_timer <= (others => '0');
1182             elsif (f_err0_timer > k_timeout) then
1183                 f_verr0_state <= s_verr0_err1;
1184                 f_err0_timer <= (others => '0');
1185             else
1186                 f_verr0_state <= s_verr0_11;
1187                 f_err0_timer <= f_err0_timer + 1;
1188             end if;
1189         -- If processor 2 is wrong
1190     else

```

```

1189         if (iMEM_READ0 = '0') and (iMEM_READ1
= '0') then
1191             f_verr0_state <= s_verr0_12;
1192             f_err0_timer <= (others => '0');
1193         elsif (f_err0_timer > k_timeout) then
1194             f_verr0_state <= s_verr0_err1;
1195             f_err0_timer <= (others => '0');
1196         else
1197             f_verr0_state <= s_verr0_11;
1198             f_err0_timer <= f_err0_timer + 1;
1199         end if;
1200     end if;
1201
1202     —Wait for incorrect processor to request next
instruction
1203     when s_verr0_12 =>
1204         — If processor 0 is wrong
1205         if (f_wrong0 = '1') then
1206             if (iMEM_READ0 = '1') then
1207                 f_verr0_state <= s_verr0_13;
1208                 f_err0_timer <= (others => '0');
1209             elsif (f_err0_timer > k_timeout) then
1210                 f_verr0_state <= s_verr0_err1;
1211                 f_err0_timer <= (others => '0');
1212             else
1213                 f_verr0_state <= s_verr0_12;
1214                 f_err0_timer <= f_err0_timer + 1;
1215             end if;
1216         — If processor 1 is wrong
1217         elsif (f_wrong1 = '1') then
1218             if (iMEM_READ1 = '1') then
1219                 f_verr0_state <= s_verr0_13;
1220                 f_err0_timer <= (others => '0');
1221             elsif (f_err0_timer > k_timeout) then
1222                 f_verr0_state <= s_verr0_err1;
1223                 f_err0_timer <= (others => '0');
1224             else
1225                 f_verr0_state <= s_verr0_12;
1226                 f_err0_timer <= f_err0_timer + 1;
1227             end if;
1228         — If processor 2 is wrong
1229         else
1230             if (iMEM_READ2 = '1') then

```

```

1231         f_verr0_state <= s_verr0_13;
1232         f_err0_timer <= (others => '0');
1233     elsif (f_err0_timer > k_timeout) then
1234         f_verr0_state <= s_verr0_err1;
1235         f_err0_timer <= (others => '0');
1236     else
1237         f_verr0_state <= s_verr0_12;
1238         f_err0_timer <= f_err0_timer + 1;
1239     end if;
1240 end if;

1241     -- Issue branch command to incorrect processor
1242     to return to the instruction call
1243     -- that caused the error. Wait for processor
1244     to acknowledge command
1245     when s_verr0_13 =>
1246     -- If processor 0 is wrong
1247     if (f_wrong0 = '1') then
1248         if (iMEM_READ0 = '0') then
1249             f_verr0_state <= s_verr0_14;
1250             f_err0_timer <= (others => '0');
1251         elsif (f_err0_timer > k_timeout) then
1252             f_verr0_state <= s_verr0_err1;
1253             f_err0_timer <= (others => '0');
1254         else
1255             f_verr0_state <= s_verr0_13;
1256             f_err0_timer <= f_err0_timer + 1;
1257         end if;
1258     -- If processor 1 is wrong
1259     elsif (f_wrong1 = '1') then
1260         if (iMEM_READ1 = '0') then
1261             f_verr0_state <= s_verr0_14;
1262             f_err0_timer <= (others => '0');
1263         elsif (f_err0_timer > k_timeout) then
1264             f_verr0_state <= s_verr0_err1;
1265             f_err0_timer <= (others => '0');
1266         else
1267             f_verr0_state <= s_verr0_13;
1268             f_err0_timer <= f_err0_timer + 1;
1269         end if;
1270     -- If processor 2 is wrong
1271     else
1272         if (iMEM_READ2 = '0') then

```

```

1271         f_verr0_state <= s_verr0_14;
1272         f_err0_timer <= (others => '0');
1273     elsif (f_err0_timer > k_timeout) then
1274         f_verr0_state <= s_verr0_err1;
1275         f_err0_timer <= (others => '0');
1276     else
1277         f_verr0_state <= s_verr0_13;
1278         f_err0_timer <= f_err0_timer + 1;
1279     end if;
1280 end if;
1281
1282     -- Wait for v fsm to retake control. If
1283     f_v fsm_state does not equal s_v fsm_err0 ,
1284     -- then f_verr0_state will be set equal to
1285     s_verr0_0
1286     when s_verr0_14 =>
1287         f_verr0_state <= s_verr0_14;
1288         -- If processor 0 is wrong
1289         if (f_wrong0 = '1') then
1290             f_done_wrong0 <= '1';
1291         -- If processor 1 is wrong
1292         elsif (f_wrong1 = '1') then
1293             f_done_wrong1 <= '1';
1294         -- If processor 2 is wrong
1295         else
1296             f_done_wrong2 <= '1';
1297         end if;
1298
1299         -- This happens when the two processors
1300         assumed to be correct have a data mismatch
1301         -- between their registers when copying data
1302         to the incorrect processor
1303         when s_verr0_err1 =>
1304             f_verr0_state <= s_verr0_err1; -- This
1305             will be overridden when v fsm transitions out of s_v fsm_err0
1306             state .
1307
1308         -- This should never happen
1309         when others =>
1310             f_verr0_state <= s_verr0_0;
1311     end case;
1312     -- Stay in s_verr0_0 state until a single processor
1313     error occurs

```

```

1307         else
1309             f_verr0_state <= s_verr0_0;
1311             f_done_wrong0 <= '0';
1313             f_done_wrong1 <= '0';
1315             f_done_wrong2 <= '0';
1317             f_err0_count <= (others => '0');
1319             f_err0_timer <= (others => '0');
1321         end if;
1323     end if;
1325 end process err0;
1327
1329 err1: process(i_clk , i_reset , f_vfsm_state)
1331 begin
1333     if (i_reset = '1') then
1335         f_verr1_state <= s_verr1_0;
1337         f_done_all_wrong <= '0';
1339         f_err1_count <= (others => '0');
1341         f_err1_timer <= (others => '0');
1343     elsif rising_edge(i_clk) then
1345         if f_vfsm_state = s_vfsm_err1 then
1347             case f_verr1_state is
1349                 -- Wait for an error to occur
1351                 when s_verr1_0 =>
1353                     f_verr1_state <= s_verr1_1;
1355
1357                 -- Reset all three processors and wait for
1359                 save location to load from memory
1361                 when s_verr1_1 =>
1363                     if iMEM_READY = '1' then
1365                         f_verr1_state <= s_verr1_2;
1367                         f_err1_save_point32 <= iMEM_OUT;
1369                     else
1371                         f_verr1_state <= s_verr1_1;
1373                     end if;
1375
1377                 -- Wait for memory ready signal to return to 0
1379                 when s_verr1_2 =>
1381                     if (iMEM_READY = '0') then
1383                         f_verr1_state <= s_verr1_2a;
1385                     else
1387                         f_verr1_state <= s_verr1_2;
1389                     end if;
1391                 end case;
1393             end if;
1395         end if;
1397     end if;
1399 end process err1;

```

```

1349      — Wait for all three processors to attempt to
      read an instruction
      when s_verr1_2a =>
1351          if (iMEM_READ0 = '1') and (iMEM_READ1 =
      '1') and (iMEM_READ2 = '1') and (iMEM_ADDRESS0 =
      iMEM_ADDRESS1) and (iMEM_ADDRESS0 = iMEM_ADDRESS2) then
              f_verr1_state <= s_verr1_3;
1353              f_err1_timer <= (others => '0');
              elsif (iMEM_READ0 = '1') and (iMEM_READ1
      = '1') and (iMEM_READ2 = '1') then
1355                  f_verr1_state <= s_verr1_0;
                  f_err1_timer <= (others => '0');
1357                  f_err1_count <= (others => '0');
                  f_done_all_wrong <= '0';
1359                  elsif (f_err1_timer > k_timeout) then
                      f_verr1_state <= s_verr1_0;
1361                      f_err1_timer <= (others => '0');
                      f_err1_count <= (others => '0');
1363                      f_done_all_wrong <= '0';
                  else
1365                      f_verr1_state <= s_verr1_2a;
                      f_err1_timer <= f_err1_timer + 1;
1367                  end if;

1369      — Issue load commands to all three processors
      when s_verr1_3 =>
1371          if (iMEM_READ0 = '0') and (iMEM_READ1 =
      '0') and (iMEM_READ2 = '0') then
              f_verr1_state <= s_verr1_4;
1373              f_err1_timer <= (others => '0');
              elsif (f_err1_timer > k_timeout) then
1375                  f_verr1_state <= s_verr1_0;
                  f_err1_timer <= (others => '0');
1377                  f_err1_count <= (others => '0');
                  f_done_all_wrong <= '0';
1379                  else
                      f_verr1_state <= s_verr1_3;
1381                      f_err1_timer <= f_err1_timer + 1;
                  end if;

1383      — Processors attempting to load data
1385      when s_verr1_4 =>
          if (iMEM_READ0 = '1') and (iMEM_READ1 =

```

```

1387      '1') and (iMEM_READ2 = '1') and (iMEM_ADDRESS0 =
iMEM_ADDRESS1) and (iMEM_ADDRESS0 = iMEM_ADDRESS2) then
1389          f_verr1_state <= s_verr1_5;
          f_err1_timer <= (others => '0');
          elsif (iMEM_READ0 = '1') and (iMEM_READ1
= '1') and (iMEM_READ2 = '1') then
1391              f_verr1_state <= s_verr1_0;
              f_err1_timer <= (others => '0');
              f_err1_count <= (others => '0');
1393              f_done_all_wrong <= '0';
          elsif (f_err1_timer > k_timeout) then
1395              f_verr1_state <= s_verr1_0;
              f_err1_timer <= (others => '0');
              f_err1_count <= (others => '0');
1397              f_done_all_wrong <= '0';
          else
1399              f_verr1_state <= s_verr1_4;
              f_err1_timer <= f_err1_timer + 1;
1401          end if;
1403
          — Read data from memory
1405          when s_verr1_5 =>
              if (iMEM_READY = '1') then
1407                  f_verr1_state <= s_verr1_6;
                  f_err1_data <= iMEM_OUT;
1409              else
                  f_verr1_state <= s_verr1_5;
1411              end if;
1413
          — Return data from memory to processors
          when s_verr1_6 =>
1415              if (iMEM_READ0 = '0') and (iMEM_READ1 =
'0') and (iMEM_READ2 = '0') then
                  if (to_integer(f_err1_count) = 30) then
1417                      f_verr1_state <= s_verr1_6a;
                      f_err1_timer <= (others => '0');
                      f_err1_count <= (others => '0');
1419                  else
                      f_verr1_state <= s_verr1_2;
                      f_err1_timer <= (others => '0');
                      f_err1_count <= f_err1_count + 1;
1421                  end if;
1423              elsif (f_err1_timer > k_timeout) then
1425

```

```

1427         f_verr1_state <= s_verr1_0;
        f_err1_timer <= (others => '0');
        f_err1_count <= (others => '0');
1429         f_done_all_wrong <= '0';
        else
1431             f_verr1_state <= s_verr1_6;
            f_err1_timer <= f_err1_timer + 1;
1433         end if;

-- Wait for memory ready signal to return to 0
1435 when s_verr1_6a =>
1437     if (iMEM_READY = '0') then
        f_verr1_state <= s_verr1_7;
1439     else
        f_verr1_state <= s_verr1_6a;
1441     end if;

1443
-- Load desired program location from memory
1445 when s_verr1_7 =>
1447     if (iMEM_READY = '1') then
        f_verr1_state <= s_verr1_8;
        f_err1_pc <= iMEM_OUT;
1449     else
        f_verr1_state <= s_verr1_7;
1451     end if;

1453
-- Wait for memory ready signal to return to 0
1455 when s_verr1_8 =>
1457     if (iMEM_READY = '0') then
        f_verr1_state <= s_verr1_8a;
        else
1459         f_verr1_state <= s_verr1_8;
        end if;

1461
-- Wait for all processors to attempt to load
an instruction
1463     when s_verr1_8a =>
        if (iMEM_READ0 = '1') and (iMEM_READ1 =
'1') and (iMEM_READ2 = '1') and (iMEM_ADDRESS0 =
iMEM_ADDRESS1) and (iMEM_ADDRESS0 = iMEM_ADDRESS2) then
1465             f_verr1_state <= s_verr1_9;

```

```

1467         f_err1_timer <= (others => '0');
1469         elsif (iMEM_READ0 = '1') and (iMEM_READ1
= '1') and (iMEM_READ2 = '1') then
            f_verr1_state <= s_verr1_0;
            f_err1_timer <= (others => '0');
            f_err1_count <= (others => '0');
1471            f_done_all_wrong <= '0';
            elsif (f_err1_timer > k_timeout) then
1473                f_verr1_state <= s_verr1_0;
                f_err1_timer <= (others => '0');
1475                f_err1_count <= (others => '0');
                f_done_all_wrong <= '0';
1477            else
                f_verr1_state <= s_verr1_8a;
1479                f_err1_timer <= f_err1_timer + 1;
            end if;

1481
1483         — Issue branch commands to all processors
        when s_verr1_9 =>
            if (iMEM_READ0 = '0') and (iMEM_READ1 =
'0') and (iMEM_READ2 = '0') then
1485                f_verr1_state <= s_verr1_10;
                f_err1_timer <= (others => '0');
1487            elsif (f_err1_timer > k_timeout) then
                f_verr1_state <= s_verr1_0;
1489                f_err1_timer <= (others => '0');
                f_err1_count <= (others => '0');
1491                f_done_all_wrong <= '0';
            else
1493                f_verr1_state <= s_verr1_9;
                f_err1_timer <= f_err1_timer + 1;
1495            end if;

1497         — Wait for v fsm to retake control. If
f_v fsm_state does not equal s_v fsm_err1,
        — then f_verr1_state will be set equal to
s_verr1_0
1499        when s_verr1_10 =>
            f_verr1_state <= s_verr1_10;
1501            f_done_all_wrong <= '1';

1503         — This should never happen
        when others =>

```

```

1505         f_verr1_state <= s_verr1_0;
1506     end case;
1507 else
1508     f_verr1_state <= s_verr1_0;
1509     f_err1_count <= (others => '0');
1510     f_err1_timer <= (others => '0');
1511     f_done_all_wrong <= '0';
1512 end if;
1513 end if;
1514 end process err1;
1515
1516 save: process(i_clk , i_reset , f_vfsm_state)
1517 begin
1518     if (i_reset = '1') then
1519         f_vsave_state <= s_vsave_0;
1520         f_done_save <= '0';
1521         f_save_count <= (others => '0');
1522         f_save_timer <= (others => '0');
1523         f_save_current_address <= (others => '0');
1524     elsif rising_edge(i_clk) then
1525         if f_vfsm_state = s_vfsm_save then
1526             case f_vsave_state is
1527                 — Wait for save point
1528                 when s_vsave_0 =>
1529                     f_vsave_state <= s_vsave_1;
1530
1531                 — Load save point from memory
1532                 when s_vsave_1 =>
1533                     if (iMEMREADY = '1') then
1534                         f_vsave_state <= s_vsave_2;
1535                         f_save_save_point32 <= iMEM_OUT;
1536                     else
1537                         f_vsave_state <= s_vsave_1;
1538                     end if;
1539
1540                 — Wait for memory ready signal to return to 0
1541                 when s_vsave_2 =>
1542                     if (iMEMREADY = '0') then
1543                         f_vsave_state <= s_vsave_2a;
1544                     else
1545                         f_vsave_state <= s_vsave_2;
1546                     end if;
1547

```

```

1549      — Wait for all three processors to attempt to
      read an instruction
      when s_vsave_2a =>
        if (iMEM_READ0 = '1') and (iMEM_READ1 =
1551 '1') and (iMEM_READ2 = '1') and (iMEM_ADDRESS0 =
iMEM_ADDRESS1) and (iMEM_ADDRESS1 = iMEM_ADDRESS2) then
          f_vsave_state <= s_vsave_3;
          f_save_timer <= (others => '0');
1553          f_save_current_address <= iMEM_ADDRESS0
;
          elsif (iMEM_READ0 = '1') and (iMEM_READ1
1555 = '1') and (iMEM_READ2 = '1') then
            f_vsave_state <= s_vsave_err1;
            f_save_count <= (others => '0');
            f_save_timer <= (others => '0');
1557          elsif (f_save_timer > k_timeout) then
            f_vsave_state <= s_vsave_err1;
            f_save_count <= (others => '0');
            f_save_timer <= (others => '0');
1559          else
            f_vsave_state <= s_vsave_2a;
            f_save_timer <= f_save_timer + 1;
1561          end if;
1563
1565      — Issue store word commands to all three
1567 processors to store the current register to memory
      when s_vsave_3 =>
1569        if (iMEM_READ0 = '0') and (iMEM_READ1 =
'0') and (iMEM_READ2 = '0') then
          f_vsave_state <= s_vsave_4;
          f_save_timer <= (others => '0');
1571        elsif (f_save_timer > k_timeout) then
          f_vsave_state <= s_vsave_err1;
          f_save_count <= (others => '0');
          f_save_timer <= (others => '0');
1573        else
          f_vsave_state <= s_vsave_3;
          f_save_timer <= f_save_timer + 1;
1575        end if;
1577
1579      — Wait for all three processors to attempt to
1581 store the current register to memory and determine if
      — at least two of the three processors agree

```

```

on the value of the current register
1583         when s_vsave_4 =>
            if (iMEM_WRITE0 = '1') and (iMEM_WRITE1 =
1585             '1') and (iMEM_WRITE2 = '1') then
                if (iMEM_IN0 = iMEM_IN1) or (iMEM_IN1
= iMEM_IN2) or (iMEM_IN0 = iMEM_IN2) then
                    f_vsave_state <= s_vsave_5;
1587                 else
                    f_vsave_state <= s_vsave_err1;
1589                 f_save_count <= (others => '0');
                end if;
                f_save_timer <= (others => '0');
1591            elsif (f_save_timer > k_timeout) then
                f_vsave_state <= s_vsave_err1;
1593                f_save_count <= (others => '0');
                f_save_timer <= (others => '0');
1595            else
                f_vsave_state <= s_vsave_4;
1597                f_save_timer <= f_save_timer + 1;
1599            end if;

-- Store the current register data to memory
1601        when s_vsave_5 =>
            if (iMEM_READY = '1') then
1603                f_vsave_state <= s_vsave_6;
            else
1605                f_vsave_state <= s_vsave_5;
            end if;

1607

-- Wait for MIPS processors write signals to
1609        return to 0
        when s_vsave_6 =>
            if (iMEM_WRITE0 = '0') and (iMEM_WRITE1 =
1611             '0') and (iMEM_WRITE2 = '0') then
                if (to_integer(f_save_count) = 30) then
1613                    f_vsave_state <= s_vsave_6a;
                    f_save_timer <= (others => '0');
1615                    f_save_count <= (others => '0');
                else
                    f_vsave_state <= s_vsave_2;
1617                    f_save_timer <= (others => '0');
                    f_save_count <= f_save_count + 1;
1619                end if;
            end if;

```

```

1621         elsif (f_save_timer > k_timeout) then
1622             f_vsave_state <= s_vsave_err1;
1623             f_save_timer <= (others => '0');
1624             f_save_count <= (others => '0');
1625         else
1626             f_vsave_state <= s_vsave_6;
1627             f_save_timer <= f_save_timer + 1;
1628         end if;
1629
1630 — Wait for memory ready signal to return to 0
1631 when s_vsave_6a =>
1632     if (iMEM_READY = '0') then
1633         f_vsave_state <= s_vsave_7;
1634     else
1635         f_vsave_state <= s_vsave_6a;
1636     end if;
1637
1638 — Write the updated PC address to memory
1639 when s_vsave_7 =>
1640     if (iMEM_READY = '1') then
1641         f_vsave_state <= s_vsave_8;
1642     else
1643         f_vsave_state <= s_vsave_7;
1644     end if;
1645
1646 — Wait for memory iMEM_READY to return to 0
1647 when s_vsave_8 =>
1648     if (iMEM_READY = '0') then
1649         f_vsave_state <= s_vsave_9;
1650     else
1651         f_vsave_state <= s_vsave_8;
1652     end if;
1653
1654 — Update the location of the save point
1655 when s_vsave_9 =>
1656     if (iMEM_READY = '1') then
1657         f_vsave_state <= s_vsave_10;
1658     else
1659         f_vsave_state <= s_vsave_9;
1660     end if;
1661
1662 — Wait for memory ready signal to return to 0
1663 when s_vsave_10 =>

```

```

1665         if (iMEM_READY = '0') then
1666             f_vsavestate <= s_vsavestate_10a;
1667         else
1668             f_vsavestate <= s_vsavestate_10;
1669         end if;

1670         — Wait for all three processors to attempt to
1671         read an instruction
1672         when s_vsavestate_10a =>
1673             if (iMEM_READ0 = '1') and (iMEM_READ1 =
1674                 '1') and (iMEM_READ2 = '1') and (iMEM_ADDRESS0 =
1675                 iMEM_ADDRESS1) and (iMEM_ADDRESS1 = iMEM_ADDRESS2) then
1676                 f_vsavestate <= s_vsavestate_11;
1677                 f_save_current_address <= iMEM_ADDRESS0
1678             ;
1679             f_save_timer <= (others => '0');
1680             elsif (iMEM_READ0 = '1') and (iMEM_READ1
1681                 = '1') and (iMEM_READ2 = '1') then
1682                 f_vsavestate <= s_vsavestate_err1;
1683                 f_save_count <= (others => '0');
1684                 f_save_timer <= (others => '0');
1685             elsif (f_save_timer > k_timeout) then
1686                 f_vsavestate <= s_vsavestate_err1;
1687                 f_save_count <= (others => '0');
1688                 f_save_timer <= (others => '0');
1689             else
1690                 f_vsavestate <= s_vsavestate_10a;
1691                 f_save_timer <= f_save_timer + 1;
1692             end if;

1693         — Issue branch commands to all three
1694         processors to return to the previous processing location
1695         when s_vsavestate_11 =>
1696             if (iMEM_READ0 = '0') and (iMEM_READ1 =
1697                 '0') and (iMEM_READ2 = '0') then
1698                 f_vsavestate <= s_vsavestate_12;
1699                 f_save_timer <= (others => '0');
1700             elsif (f_save_timer > k_timeout) then
1701                 f_vsavestate <= s_vsavestate_err1;
1702                 f_save_count <= (others => '0');
1703                 f_save_timer <= (others => '0');
1704             else
1705                 f_vsavestate <= s_vsavestate_11;

```

```

1701         f_save_timer <= f_save_timer + 1;
1702     end if;

1703     -- Wait for FSM to resume control
1704     when s_vsav_12 =>
1705         f_vsav_state <= s_vsav_12;
1706         f_done_save <= '1';

1707     -- Error occurred during save
1708     when s_vsav_err1 =>
1709         f_vsav_state <= s_vsav_err1;

1710     when others =>

1711     end case;

1712 else
1713     f_vsav_state <= s_vsav_0;
1714     f_save_count <= (others => '0');
1715     f_save_timer <= (others => '0');
1716     f_done_save <= '0';
1717 end if;

1718 end if;
1719 end process save;

1720
1721 voter_output_fsm: process(i_clk , i_reset , f_vfsm_state ,
1722 f_verr0_state , f_verr1_state , f_vsav_state)
1723 begin
1724     if (i_reset = '1') then
1725         f_MEM_READ <= '0';
1726         f_MEM_WRITE <= '0';
1727         f_MEM_IN <= (others => '0');
1728         f_MEM_ADDRESS <= (others => '0');
1729         f_MEM_OUT0 <= (others => '0');
1730         f_MEM_OUT1 <= (others => '0');
1731         f_MEM_OUT2 <= (others => '0');
1732         f_MEM_READY0 <= '0';
1733         f_MEM_READY1 <= '0';
1734         f_MEM_READY2 <= '0';
1735         f_RESET0 <= '0';
1736         f_RESET1 <= '0';
1737         f_RESET2 <= '0';
1738         f_NEXT_INSTR <= '0';
1739         f_TMR_ERROR <= '0';
1740     end if;

```

```

1743         f_Err_Override <= '0';
1744     elsif rising_edge(i_clk) then
1745         case f_vfsm_state is
1746             — Determine when all three processors are
attempting to read or write.
1747             when s_vfsm_0 =>
1748                 — Do nothing as this is the idle state, but
ensure all outputs are suppressed
1749                 f_MEM_READ <= '0';
1750                 f_MEM_WRITE <= '0';
1751                 f_MEM_IN <= (others => '0');
1752                 f_MEM_ADDRESS <= (others => '0');
1753                 f_MEM_READY0 <= '0';
1754                 f_MEM_READY1 <= '0';
1755                 f_MEM_READY2 <= '0';
1756                 f_RESET0 <= '0';
1757                 f_RESET1 <= '0';
1758                 f_RESET2 <= '0';
1759                 f_TMR_ERROR <= '0';
1760                 f_NEXT_INSTR <= '1';
1761                 f_Err_Override <= '0';
1762
1763             — Determine whether this is a read or write
operation or if an error has occurred with one or more
processors
1764             when s_vfsm_1 =>
1765                 — Do nothing while the vote is determined
1766                 f_TMR_ERROR <= '0';
1767                 f_NEXT_INSTR <= '0';
1768
1769             — Request data from memory by sending address
and awaiting response from memory.
1770             when s_vfsm_2 =>
1771                 f_MEM_READ <= i_MEM_READ0;
1772                 f_MEM_ADDRESS <= i_MEM_ADDRESS0;
1773                 f_TMR_ERROR <= '0';
1774                 f_NEXT_INSTR <= '0';
1775
1776             — Pass the instruction to all three processors
1777             when s_vfsm_3 =>
1778                 f_MEM_READ <= '0';
1779                 f_MEM_ADDRESS <= (others => '0');
1780                 f_MEM_OUT0 <= f_data;

```

```

1781         f_MEM_OUT1 <= f_data;
1782         f_MEM_OUT2 <= f_data;
1783         f_MEM_READY0 <= '1';
1784         f_MEM_READY1 <= '1';
1785         f_MEM_READY2 <= '1';
1786         f_TMR_ERROR <= '0';
1787         f_NEXT_INSTR <= '0';
1788
1789     when s_vfsm_3a =>
1790         f_MEM_READ <= '0';
1791         f_MEM_ADDRESS <= (others => '0');
1792         f_MEM_OUT0 <= f_data;
1793         f_MEM_OUT1 <= f_data;
1794         f_MEM_OUT2 <= f_data;
1795         f_MEM_READY0 <= '0';
1796         f_MEM_READY1 <= '0';
1797         f_MEM_READY2 <= '0';
1798         f_TMR_ERROR <= '0';
1799         f_NEXT_INSTR <= '0';
1800
1801         — This is a load word operation. Wait for all
1802         three processors to request data from memory
1803         when s_vfsm_4 =>
1804             f_MEM_READY0 <= '0';
1805             f_MEM_READY1 <= '0';
1806             f_MEM_READY2 <= '0';
1807             f_TMR_ERROR <= '0';
1808             f_NEXT_INSTR <= '0';
1809
1810         — Check to make sure all three processors agree.
1811         If not, at least one processor is in error.
1812         when s_vfsm_5 =>
1813             — Do nothing while the vote is determined
1814             f_TMR_ERROR <= '0';
1815             f_NEXT_INSTR <= '0';
1816
1817         — Inform correct processors that write operation
1818         completed successfully before continuing to ERR0
1819         when s_vfsm_5a =>
1820             if (f_wrong0 = '1') then
1821                 f_MEM_READY1 <= '1';
1822                 f_MEM_READY2 <= '1';
1823             elsif (f_wrong1 = '1') then

```

```

1821         f_MEM_READY0 <= '1';
1822         f_MEM_READY2 <= '1';
1823     elsif (f_wrong2 = '1') then
1824         f_MEM_READY0 <= '1';
1825         f_MEM_READY1 <= '1';
1826     end if;
1827     f_TMR_ERROR <= '0';
1828     f_NEXT_INSTR <= '0';

1829     — Request data from memory by sending address
and awaiting response from memory.
1830     when s_vfsm_6 =>
1831         f_MEM_READ <= i_MEM_READ0;
1832         f_MEM_ADDRESS <= i_MEM_ADDRESS0;
1833         f_TMR_ERROR <= '0';
1834         f_NEXT_INSTR <= '0';

1835     — Pass data back to all three processors
1836     when s_vfsm_7 =>
1837         f_MEM_READ <= '0';
1838         f_MEM_ADDRESS <= (others => '0');
1839         f_MEM_OUT0 <= f_data;
1840         f_MEM_OUT1 <= f_data;
1841         f_MEM_OUT2 <= f_data;
1842         f_MEM_READY0 <= '1';
1843         f_MEM_READY1 <= '1';
1844         f_MEM_READY2 <= '1';
1845         f_TMR_ERROR <= '0';
1846         f_NEXT_INSTR <= '0';

1847

1848     when s_vfsm_7a =>
1849         f_MEM_READ <= '0';
1850         f_MEM_ADDRESS <= (others => '0');
1851         f_MEM_OUT0 <= f_data;
1852         f_MEM_OUT1 <= f_data;
1853         f_MEM_OUT2 <= f_data;
1854         f_MEM_READY0 <= '0';
1855         f_MEM_READY1 <= '0';
1856         f_MEM_READY2 <= '0';
1857         f_TMR_ERROR <= '0';
1858         f_NEXT_INSTR <= '0';

1859

1860     — This is a store word operation. Wait for at

```

```

least two of three processors to send data to memory
when s_vfsm_8 =>
1863     f_MEM_READY0 <= '0';
1865     f_MEM_READY1 <= '0';
1867     f_MEM_READY2 <= '0';
1869     f_TMR_ERROR <= '0';
1871     f_NEXT_INSTR <= '0';

    — Check to make sure all three processors agree.
    If not, at least one processor is in error.
    when s_vfsm_9 =>
1873         — Do nothing but wait for vote outcome
1875         f_TMR_ERROR <= '0';
1877         f_NEXT_INSTR <= '0';

    — Inform correct processors that write operation
    completed successfully before continuing to ERR0
    when s_vfsm_9a =>
1879         if (f_wrong0 = '1') then
1881             f_MEM_READY1 <= '1';
1883             f_MEM_READY2 <= '1';
1885             elsif (f_wrong1 = '1') then
1887                 f_MEM_READY0 <= '1';
1889                 f_MEM_READY2 <= '1';
1891                 elsif (f_wrong2 = '1') then
1893                     f_MEM_READY0 <= '1';
1895                     f_MEM_READY1 <= '1';
1897                     end if;
1899                     f_TMR_ERROR <= '0';
1901                     f_NEXT_INSTR <= '0';

    — Write data to memory by sending address and
    data and awaiting response from memory.
    when s_vfsm_10 =>
1903         f_MEM_WRITE <= i_MEM_WRITE0;
1905         f_MEM_ADDRESS <= i_MEM_ADDRESS0;
1907         f_MEM_IN <= i_MEM_IN0;
1909         f_TMR_ERROR <= '0';
1911         f_NEXT_INSTR <= '0';

    — Pass MEMREADY signal back to all processors
    when s_vfsm_11 =>
1913         f_MEM_WRITE <= '0';

```

```

1901         f_MEM_ADDRESS <= (others => '0');
1902         f_MEM_READY0 <= '1';
1903         f_MEM_READY1 <= '1';
1904         f_MEM_READY2 <= '1';
1905         f_TMR_ERROR <= '0';
1906         f_NEXT_INSTR <= '0';
1907
1908     when s_vfsm_12 =>
1909         f_MEM_WRITE <= '0';
1910         f_MEM_ADDRESS <= (others => '0');
1911         f_MEM_READY0 <= '0';
1912         f_MEM_READY1 <= '0';
1913         f_MEM_READY2 <= '0';
1914         f_TMR_ERROR <= '0';
1915         f_NEXT_INSTR <= '0';
1916
1917     -- Enter this state when one of three processors
are wrong
1918     when s_vfsm_err0 =>
1919         f_Err_Override <= '1';
1920         f_TMR_ERROR <= '1';
1921         f_NEXT_INSTR <= '0';
1922         case f_verr0_state is
1923             -- Waiting for an error to occur in one of
the processors
1924             when s_verr0_0 =>
1925                 f_MEM_READY0 <= '0';
1926                 f_MEM_READY1 <= '0';
1927                 f_MEM_READY2 <= '0';
1928                 -- Do nothing
1929
1930             -- Reset the processor that is in error
1931             when s_verr0_1 =>
1932                 -- If processor 0 is wrong
1933                 if f_wrong0 = '1' then
1934                     f_RESET0 <= '1';
1935                 -- If processor 1 is wrong
1936                 elsif f_wrong1 = '1' then
1937                     f_RESET1 <= '1';
1938                 -- If processor 2 is wrong
1939                 else
1940                     f_RESET2 <= '1';
1941                 end if;

```

```

1943         f_MEM_READY0 <= '0';
1944         f_MEM_READY1 <= '0';
1945         f_MEM_READY2 <= '0';
1946
1947         --| Copy data from correct processors to
1948         incorrect processor one register at a time
1949         -- Wait for correct processors to attempt
1950         to retrieve an instruction from memory'
1951         when s_verr0_2 =>
1952             -- If processor 0 is wrong
1953             if f_wrong0 = '1' then
1954                 f_RESET0 <= '0';
1955             -- If processor 1 is wrong
1956             elsif f_wrong1 = '1' then
1957                 f_RESET1 <= '0';
1958             -- If processor 2 is wrong
1959             else
1960                 f_RESET2 <= '0';
1961             end if;
1962             f_MEM_READY0 <= '0';
1963             f_MEM_READY1 <= '0';
1964             f_MEM_READY2 <= '0';
1965
1966         -- Issue store word commands to the correct
1967         processors
1968         when s_verr0_3 =>
1969             -- If processor 0 is wrong
1970             if f_wrong0 = '1' then
1971                 f_MEM_OUT1 <= "10101100000" &
1972                 std_logic_vector(f_err0_count+1) & "0000000000000000";
1973                 f_MEM_OUT2 <= "10101100000" &
1974                 std_logic_vector(f_err0_count+1) & "0000000000000000";
1975                 f_MEM_READY1 <= '1';
1976                 f_MEM_READY2 <= '1';
1977             -- If processor 1 is wrong
1978             elsif f_wrong1 = '1' then
1979                 f_MEM_OUT0 <= "10101100000" &
1980                 std_logic_vector(f_err0_count+1) & "0000000000000000";
1981                 f_MEM_OUT2 <= "10101100000" &
1982                 std_logic_vector(f_err0_count+1) & "0000000000000000";
1983                 f_MEM_READY0 <= '1';
1984                 f_MEM_READY2 <= '1';
1985             -- If processor 2 is wrong

```

```

else
    f_MEM_OUT0 <= "10101100000" &
std_logic_vector(f_err0_count+1) & "0000000000000000";
    f_MEM_OUT1 <= "10101100000" &
std_logic_vector(f_err0_count+1) & "0000000000000000";
    f_MEM_READY0 <= '1';
    f_MEM_READY1 <= '1';
end if;

-- Wait for correct processors to issue
store word requests
when s_verr0_4 =>
    -- If processor 0 is wrong
    if f_wrong0 = '1' then
        f_MEM_READY1 <= '0';
        f_MEM_READY2 <= '0';
    -- If processor 1 is wrong
    elsif f_wrong1 = '1' then
        f_MEM_READY0 <= '0';
        f_MEM_READY2 <= '0';
    -- If processor 2 is wrong
    else
        f_MEM_READY0 <= '0';
        f_MEM_READY1 <= '0';
    end if;

    -- Check that data from the correct
processors is a match
when s_verr0_5 =>
    -- If processor 0 is wrong
    if f_wrong0 = '1' then
        f_MEM_READY1 <= '1';
        f_MEM_READY2 <= '1';
    -- If processor 1 is wrong
    elsif f_wrong1 = '1' then
        f_MEM_READY0 <= '1';
        f_MEM_READY2 <= '1';
    -- If processor 2 is wrong
    else
        f_MEM_READY0 <= '1';
        f_MEM_READY1 <= '1';
    end if;

```

```

2017         — Wait for incorrect processor to attempt
to retrieve an instruction from memory
        when s_verr0_6 =>
2019             — If processor 0 is wrong
            if f_wrong0 = '1' then
2021                 f_MEM_READY1 <= '0';
                f_MEM_READY2 <= '0';
2023             — If processor 1 is wrong
            elsif f_wrong1 = '1' then
2025                 f_MEM_READY0 <= '0';
                f_MEM_READY2 <= '0';
2027             — If processor 2 is wrong
            else
2029                 f_MEM_READY0 <= '0';
                f_MEM_READY1 <= '0';
2031             end if;

2033         — Issue load word command to the incorrect
processor
        when s_verr0_7 =>
2035             — If processor 0 is wrong
            if f_wrong0 = '1' then
2037                 f_MEM_OUT0 <= "10001100000" &
std_logic_vector(f_err0_count+1) & "0000000000000000";
                f_MEM_READY0 <= '1';
2039             — If processor 1 is wrong
            elsif f_wrong1 = '1' then
2041                 f_MEM_OUT1 <= "10001100000" &
std_logic_vector(f_err0_count+1) & "0000000000000000";
                f_MEM_READY1 <= '1';
2043             — If processor 2 is wrong
            else
2045                 f_MEM_OUT2 <= "10001100000" &
std_logic_vector(f_err0_count+1) & "0000000000000000";
                f_MEM_READY2 <= '1';
2047             end if;

2049         — Wait for incorrect processor to attempt
to load the requested data from memory
        when s_verr0_8 =>
2051             — If processor 0 is wrong
            if f_wrong0 = '1' then
2053                 f_MEM_READY0 <= '0';

```

```

2055      -- If processor 1 is wrong
2056      elsif f_wrong1 = '1' then
2057          f_MEM_READY1 <= '0';
2058      -- If processor 2 is wrong
2059      else
2060          f_MEM_READY2 <= '0';
2061      end if;
2062
2063      -- Return desired data to the incorrect
processor
2064      when s_verr0_9 =>
2065          -- If processor 0 is wrong
2066          if f_wrong0 = '1' then
2067              f_MEM_OUT0 <= f_err0_data;
2068              f_MEM_READY0 <= '1';
2069          -- If processor 1 is wrong
2070          elsif f_wrong1 = '1' then
2071              f_MEM_OUT1 <= f_err0_data;
2072              f_MEM_READY1 <= '1';
2073          -- If processor 2 is wrong
2074          else
2075              f_MEM_OUT2 <= f_err0_data;
2076              f_MEM_READY2 <= '1';
2077          end if;
2078
2079      --| Return processor's program counters to
the point at which the error was encountered
2080      --Wait for correct processors to request
next instruction
2081      when s_verr0_10 =>
2082          -- If processor 0 is wrong
2083          if f_wrong0 = '1' then
2084              f_MEM_READY0 <= '0';
2085          -- If processor 1 is wrong
2086          elsif f_wrong1 = '1' then
2087              f_MEM_READY1 <= '0';
2088          -- If processor 2 is wrong
2089          else
2090              f_MEM_READY2 <= '0';
2091          end if;
2092
2093      -- Issue branch command to correct
processors to return to the instruction call

```

```

2093         — that caused the error.  Wait for
processors to acknowledge command
        when s_verr0_11 =>
2095         — If processor 0 is wrong
        if f_wrong0 = '1' then
2097             f_MEM_OUT1 <= "0001000000000000" &
w_correct_branch_distance16;
                f_MEM_OUT2 <= "0001000000000000" &
w_correct_branch_distance16;
2099             f_MEM_READY1 <= '1';
                f_MEM_READY2 <= '1';
2101         — If processor 1 is wrong
        elsif f_wrong1 = '1' then
2103             f_MEM_OUT0 <= "0001000000000000" &
w_correct_branch_distance16;
                f_MEM_OUT2 <= "0001000000000000" &
w_correct_branch_distance16;
2105             f_MEM_READY0 <= '1';
                f_MEM_READY2 <= '1';
2107         — If processor 2 is wrong
        else
2109             f_MEM_OUT0 <= "0001000000000000" &
w_correct_branch_distance16;
                f_MEM_OUT1 <= "0001000000000000" &
w_correct_branch_distance16;
2111             f_MEM_READY0 <= '1';
                f_MEM_READY1 <= '1';
2113         end if;

2115         —Wait for incorrect processor to request
next instruction
        when s_verr0_12 =>
2117         — If processor 0 is wrong
        if f_wrong0 = '1' then
2119             f_MEM_READY1 <= '0';
                f_MEM_READY2 <= '0';
2121         — If processor 1 is wrong
        elsif f_wrong1 = '1' then
2123             f_MEM_READY0 <= '0';
                f_MEM_READY2 <= '0';
2125         — If processor 2 is wrong
        else
2127             f_MEM_READY0 <= '0';

```

```

2129         fMEM_READY1 <= '0';
        end if;

2131         -- Issue branch command to incorrect
processor to return to the instruction call
        -- that caused the error. Wait for
processor to acknowledge command
2133         when s_verr0_13 =>
            -- If processor 0 is wrong
2135             if f_wrong0 = '1' then
                fMEM_OUT0 <= "0001000000000000" &
w_incorrect_branch_distance16;
2137                 fMEM_READY0 <= '1';
            -- If processor 1 is wrong
2139             elsif f_wrong1 = '1' then
                fMEM_OUT1 <= "0001000000000000" &
w_incorrect_branch_distance16;
2141                 fMEM_READY1 <= '1';
            -- If processor 2 is wrong
2143             else
                fMEM_OUT2 <= "0001000000000000" &
w_incorrect_branch_distance16;
2145                 fMEM_READY2 <= '1';
            end if;

2147         -- Wait for v fsm to retake control. If
f_v fsm_state does not equal s_v fsm_err0 ,
2149         -- then f_verr0_state will be set equal to
s_verr0_0

        when s_verr0_14 =>
            -- If processor 0 is wrong
2151             if f_wrong0 = '1' then
                fMEM_READY0 <= '0';
            -- If processor 1 is wrong
2153             elsif f_wrong1 = '1' then
                fMEM_READY1 <= '0';
            -- If processor 2 is wrong
2155             else
                fMEM_READY2 <= '0';
2157             end if;

2159         -- This should never happen
        when others =>
2163

```

```

2165         end case;
2166         — Enter this state when all three processors are
wrong
2167         when s_vfsm_err1 =>
2168             f_Err_Override <= '1';
2169             f_TMR_ERROR <= '1';
2170             f_NEXT_INSTR <= '0';
2171             case f_verr1_state is
2172                 — Wait for an error to occur
2173                 when s_verr1_0 =>
2174                     f_MEM_READY0 <= '0';
2175                     f_MEM_READY1 <= '0';
2176                     f_MEM_READY2 <= '0';
2177
2178                 — Reset all three processors and wait for
save location to load from memory
2179                 when s_verr1_1 =>
2180                     f_RESET0 <= '1';
2181                     f_RESET1 <= '1';
2182                     f_RESET2 <= '1';
2183                     f_MEM_ADDRESS <= std_logic_vector(
unsigned(k_mem_location) + unsigned(k_256_32));—
("0000000000000000000000000000" & std_logic_vector(
f_err1_count) & "00");
2184                     f_MEM_READ <= '1';
2185                     f_MEM_READY0 <= '0';
2186                     f_MEM_READY1 <= '0';
2187                     f_MEM_READY2 <= '0';
2188
2189                 — Wait for memory ready signal to return
to 0
2190                 when s_verr1_2 =>
2191                     f_RESET0 <= '0';
2192                     f_RESET1 <= '0';
2193                     f_RESET2 <= '0';
2194                     f_MEM_READY0 <= '0';
2195                     f_MEM_READY1 <= '0';
2196                     f_MEM_READY2 <= '0';
2197                     f_MEM_ADDRESS <= (others => '0');
2198                     f_MEM_READ <= '0';
2199
2200                 — Wait for all three processors to attempt

```

```

to read an instruction
2201         when s_verr1_2a =>
                f_RESET0 <= '0';
2203         f_RESET1 <= '0';
                f_RESET2 <= '0';
2205         f_MEM_READY0 <= '0';
                f_MEM_READY1 <= '0';
2207         f_MEM_READY2 <= '0';
                f_MEM_ADDRESS <= (others => '0');
2209         f_MEM_READ <= '0';

-- Issue load commands to all three
2211 processors
        when s_verr1_3 =>
2213             if (f_err1_save_point32(0) = '0') then
                    f_MEM_OUT0 <= "10001100000" &
std_logic_vector(f_err1_count+1) & std_logic_vector(
unsigned(k_mem_location(15 downto 0)) + unsigned("
000000000" & std_logic_vector(f_err1_count) & "00"));
2215             f_MEM_OUT1 <= "10001100000" &
std_logic_vector(f_err1_count+1) & std_logic_vector(
unsigned(k_mem_location(15 downto 0)) + unsigned("
000000000" & std_logic_vector(f_err1_count) & "00"));
                    f_MEM_OUT2 <= "10001100000" &
std_logic_vector(f_err1_count+1) & std_logic_vector(
unsigned(k_mem_location(15 downto 0)) + unsigned("
000000000" & std_logic_vector(f_err1_count) & "00"));
2217             else
                    f_MEM_OUT0 <= "10001100000" &
std_logic_vector(f_err1_count+1) & std_logic_vector(
unsigned(k_mem_location(15 downto 0)) + unsigned(k_128_16)
+ unsigned("000000000" & std_logic_vector(f_err1_count) &
"00"));
2219             f_MEM_OUT1 <= "10001100000" &
std_logic_vector(f_err1_count+1) & std_logic_vector(
unsigned(k_mem_location(15 downto 0)) + unsigned(k_128_16)
+ unsigned("000000000" & std_logic_vector(f_err1_count) &
"00"));
                    f_MEM_OUT2 <= "10001100000" &
std_logic_vector(f_err1_count+1) & std_logic_vector(
unsigned(k_mem_location(15 downto 0)) + unsigned(k_128_16)
+ unsigned("000000000" & std_logic_vector(f_err1_count) &
"00"));

```

```

2221         end if;
2222         f_MEM_READY0 <= '1';
2223         f_MEM_READY1 <= '1';
2224         f_MEM_READY2 <= '1';
2225
2226         -- Processors attempting to load data
2227         when s_verr1_4 =>
2228             f_MEM_READY0 <= '0';
2229             f_MEM_READY1 <= '0';
2230             f_MEM_READY2 <= '0';
2231
2232         -- Read data from memory
2233         when s_verr1_5 =>
2234             f_MEM_ADDRESS <= i_MEM_ADDRESS0;
2235             f_MEM_READ <= '1';
2236
2237         -- Return data from memory to processors
2238         when s_verr1_6 =>
2239             f_MEM_ADDRESS <= (others => '0');
2240             f_MEM_READ <= '0';
2241             f_MEM_OUT0 <= f_err1_data;
2242             f_MEM_OUT1 <= f_err1_data;
2243             f_MEM_OUT2 <= f_err1_data;
2244             f_MEM_READY0 <= '1';
2245             f_MEM_READY1 <= '1';
2246             f_MEM_READY2 <= '1';
2247
2248         -- Wait for memory ready signal to return
2249         to 0
2250         when s_verr1_6a =>
2251             f_MEM_ADDRESS <= (others => '0');
2252             f_MEM_READ <= '0';
2253             f_MEM_OUT0 <= f_err1_data;
2254             f_MEM_OUT1 <= f_err1_data;
2255             f_MEM_OUT2 <= f_err1_data;
2256             f_MEM_READY0 <= '0';
2257             f_MEM_READY1 <= '0';
2258             f_MEM_READY2 <= '0';
2259
2260         -- Load desired program location from
2261         memory
2262         when s_verr1_7 =>
2263             f_MEM_READY0 <= '0';

```

```

2263         f_MEM_READY1 <= '0';
2264         f_MEM_READY2 <= '0';
2265         if (f_err1_save_point32(0) = '0') then
2266             f_MEM_ADDRESS <= std_logic_vector(
unsigned(k_mem_location) + unsigned(k_124_32));
2267         else
2268             f_MEM_ADDRESS <= std_logic_vector(
unsigned(k_mem_location) + unsigned(k_252_32));
2269         end if;
2270         f_MEM_READ <= '1';

2271         — Wait for all processors to attempt to
load an instruction
2272         when s_verr1_8 =>
2273             f_MEM_ADDRESS <= (others => '0');
2274             f_MEM_READ <= '0';

2275         — Wait for memory ready signal to return
to 0
2276         when s_verr1_8a =>
2277             f_MEM_ADDRESS <= (others => '0');
2278             f_MEM_READ <= '0';

2279         — Issue branch commands to all processors
2280         when s_verr1_9 =>
2281             f_MEM_OUT0 <= "0001000000000000" &
w_err1_branch_distance16;
2282             f_MEM_OUT1 <= "0001000000000000" &
w_err1_branch_distance16;
2283             f_MEM_OUT2 <= "0001000000000000" &
w_err1_branch_distance16;
2284             f_MEM_READY0 <= '1';
2285             f_MEM_READY1 <= '1';
2286             f_MEM_READY2 <= '1';

2287         — Wait for v fsm to retake control. If
f_v fsm_state dows not equal s_v fsm_err1,
2288         — then f_verr1_state will be set equal to
s_verr1_0
2289         when s_verr1_10 =>
2290             f_MEM_READY0 <= '0';
2291             f_MEM_READY1 <= '0';
2292             f_MEM_READY2 <= '0';

```

```

2297         — This should never happen
           when others =>

2299
           end case;

2301         — Enter this state when it is time to create a
           save point from which a recovery may take place
2303         when s_vfsm_save =>
           f_Err_Override <= '1';
2305         f_TMR_ERROR <= '0';
           f_NEXT_INSTR <= '0';
2307         case f_vsavestate is
           — Wait for save point
2309         when s_vsavestate_0 =>
           f_MEM_READY0 <= '0';
2311           f_MEM_READY1 <= '0';
           f_MEM_READY2 <= '0';

2313
           — Load save point from memory
2315         when s_vsavestate_1 =>
           f_MEM_ADDRESS <= std_logic_vector(
           unsigned(k_mem_location) + unsigned(k_256_32));
2317           f_MEM_READ <= '1';

2319
           — Wait for memory ready signal to return
           to 0

           when s_vsavestate_2 =>
2321           f_MEM_ADDRESS <= (others => '0');
           f_MEM_READ <= '0';
2323           f_MEM_READY0 <= '0';
           f_MEM_READY1 <= '0';
2325           f_MEM_READY2 <= '0';

2327
           — Wait for all three processors to attempt
           to read an instruction
           when s_vsavestate_2a =>
2329           f_MEM_ADDRESS <= (others => '0');
           f_MEM_READ <= '0';
2331           f_MEM_READY0 <= '0';
           f_MEM_READY1 <= '0';
2333           f_MEM_READY2 <= '0';

```

```

2335         — Issue store word commands to all three
processors to store the current register to memory
        when s_vsave_3 =>
2337             if f_save_save_point32(0) = '1' then
                f_MEM_OUT0 <= "10101100000" &
std_logic_vector(f_save_count+1) & std_logic_vector(
unsigned(k_mem_location(15 downto 0)) + unsigned("
000000000" & std_logic_vector(f_save_count) & "00"));
2339                f_MEM_OUT1 <= "10101100000" &
std_logic_vector(f_save_count+1) & std_logic_vector(
unsigned(k_mem_location(15 downto 0)) + unsigned("
000000000" & std_logic_vector(f_save_count) & "00"));
                f_MEM_OUT2 <= "10101100000" &
std_logic_vector(f_save_count+1) & std_logic_vector(
unsigned(k_mem_location(15 downto 0)) + unsigned("
000000000" & std_logic_vector(f_save_count) & "00"));
2341            else
                f_MEM_OUT0 <= "10101100000" &
std_logic_vector(f_save_count+1) & std_logic_vector(
unsigned(k_mem_location(15 downto 0)) + unsigned(k_128_16)
+ unsigned("000000000" & std_logic_vector(f_save_count) &
"00"));
2343                f_MEM_OUT1 <= "10101100000" &
std_logic_vector(f_save_count+1) & std_logic_vector(
unsigned(k_mem_location(15 downto 0)) + unsigned(k_128_16)
+ unsigned("000000000" & std_logic_vector(f_save_count) &
"00"));
                f_MEM_OUT2 <= "10101100000" &
std_logic_vector(f_save_count+1) & std_logic_vector(
unsigned(k_mem_location(15 downto 0)) + unsigned(k_128_16)
+ unsigned("000000000" & std_logic_vector(f_save_count) &
"00"));
2345            end if;
            f_MEM_READY0 <= '1';
2347            f_MEM_READY1 <= '1';
            f_MEM_READY2 <= '1';
2349
2351         — Wait for all three processors to attempt
to store the current register to memory and determine if
        — at least two of the three processors
agree on the value of the current register
2353         when s_vsave_4 =>

```

```

2355         f_MEM_READY0 <= '0';
2356         f_MEM_READY1 <= '0';
2357         f_MEM_READY2 <= '0';

2358     — Store the current register data to
memory
2359     when s_vsave_5 =>
2360         if (i_MEM_IN0 = i_MEM_IN1) then
2361             f_MEM_IN <= i_MEM_IN0;
2362         elsif (i_MEM_IN1 = i_MEM_IN2) then
2363             f_MEM_IN <= i_MEM_IN1;
2364         elsif (i_MEM_IN0 = i_MEM_IN2) then
2365             f_MEM_IN <= i_MEM_IN0;
2366         end if;
2367         if f_save_save_point32(0) = '1' then
2368             f_MEM_ADDRESS <= "0000000000000000" &
std_logic_vector(unsigned(k_mem_location(15 downto 0)) +
unsigned("00000000" & std_logic_vector(f_save_count) & "
00"));
2369         else
2370             f_MEM_ADDRESS <= "0000000000000000" &
std_logic_vector(unsigned(k_mem_location(15 downto 0)) +
unsigned(k_128_16) + unsigned("00000000" &
std_logic_vector(f_save_count) & "00"));
2371         end if;
2372         f_MEM_WRITE <= '1';

2373     — Wait for all three processors to return
the memory write signals to 0
2374     when s_vsave_6 =>
2375         f_MEM_IN <= (others => '0');
2376         f_MEM_ADDRESS <= (others => '0');
2377         f_MEM_WRITE <= '0';
2378         f_MEM_READY0 <= '1';
2379         f_MEM_READY1 <= '1';
2380         f_MEM_READY2 <= '1';

2381     — Wait for memory iMEM_READY to return to
0
2382     when s_vsave_6a =>
2383         f_MEM_IN <= (others => '0');
2384         f_MEM_ADDRESS <= (others => '0');
2385         f_MEM_WRITE <= '0';

```

```

2389         f_MEM_READY0 <= '0';
2391         f_MEM_READY1 <= '0';
2393         f_MEM_READY2 <= '0';

2395         -- Write the updated PC address to memory
2397         when s_vsave_7 =>
2399             f_MEM_IN <= f_save_address;
2401             if f_save_save_point32(0) = '1' then
2403                 f_MEM_ADDRESS <= std_logic_vector(
2405                     unsigned(k_mem_location) + unsigned(k_124_32));
2407             else
2409                 f_MEM_ADDRESS <= std_logic_vector(
2411                     unsigned(k_mem_location) + unsigned(k_252_32));
2413             end if;
2415             f_MEM_WRITE <= '1';
2417             f_MEM_READY0 <= '0';
2419             f_MEM_READY1 <= '0';
2421             f_MEM_READY2 <= '0';

2423         -- Wait for memory iMEM_READY to return to
2425         0

2427         when s_vsave_8 =>
2429             f_MEM_IN <= (others => '0');
2431             f_MEM_ADDRESS <= (others => '0');
2433             f_MEM_WRITE <= '0';

2435         -- Update the save point
2437         when s_vsave_9 =>
2439             if f_save_save_point32(0) = '1' then
2441                 f_MEM_IN <= (others => '0');
2443             else
2445                 f_MEM_IN <= "
2447                 11111111111111111111111111111111";
2449             end if;
2451             f_MEM_ADDRESS <= std_logic_vector(
2453                 unsigned(k_mem_location) + unsigned(k_256_32));
2455             f_MEM_WRITE <= '1';

2457         -- Wait for memory ready signal to return
2459         to 0

2461         when s_vsave_10 =>
2463             f_MEM_IN <= (others => '0');
2465             f_MEM_ADDRESS <= (others => '0');

```

```

2425         f_MEM_WRITE <= '0';

2427         — Wait for all three processors to attempt
to read an instruction
        when s_vsave_10a =>
2429             f_MEM_IN <= (others => '0');
                f_MEM_ADDRESS <= (others => '0');
2431             f_MEM_WRITE <= '0';

2433         — Issue branch commands to all three
processors to return to the previous processing location
        when s_vsave_11 =>
2435             f_MEM_OUT0 <= "0001000000000000" &
w_save_branch_distance16;
                f_MEM_OUT1 <= "0001000000000000" &
w_save_branch_distance16;
2437             f_MEM_OUT2 <= "0001000000000000" &
w_save_branch_distance16;
                f_MEM_READY0 <= '1';
2439             f_MEM_READY1 <= '1';
                f_MEM_READY2 <= '1';

2441         — Wait for FSM to resume control
2443         when s_vsave_12 =>
                f_MEM_READY0 <= '0';
2445             f_MEM_READY1 <= '0';
                f_MEM_READY2 <= '0';

2447         when others =>

2449             end case;

2451         — This should never happen
2453         when others =>
                f_TMR_ERROR <= '0';
2455             f_NEXT_INSTR <= '0';

                end case;
2457         end if;
        end process voter_output_fsm;

2459 end a_TMR_Voter_Test1001;

```

Listing A.1. TMR_Voter.vhd Code

Appendix B. Version History

- Version 1.4
 - Converted document to AFIT Report Format
- Version 1.3
 - Added hyperlinks to Table of Contents and table references to simplify document navigation
- Version 1.2
 - Minor corrections and updates
- Version 1.1
 - Updated state ERR1_2 outputs in Table 7
 - Previous version had special states for load word instructions, but not store word instructions. Update adds special states for store word instructions so that a store word instruction is read from memory and passed onto the Basic MIPS processors in states 1-3, then in states 8-11 wait for the write enable signal from the processors and write the data to memory while performing the required error checking
 - When loading a word from memory to a general purpose register in Basic MIPS in the previous version, if an error was encountered, the error address was chosen to be the memory address of the word to be loaded from memory into the Basic MIPS general purpose register. This is incorrect. The location of the load word instruction must be used. This has been corrected in this version
- Version 1.0

– Original Document.

Bibliography

1. N. S. Hamilton, “Basic MIPS Architecture Version 1.4,” Jul 2019.

REPORT DOCUMENTATION PAGE					<i>Form Approved</i> <i>OMB No. 0704-0188</i>	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.						
1. REPORT DATE (DD-MM-YYYY) 12-09-2019		2. REPORT TYPE Technical Report		3. DATES COVERED (From — To) Sept 2016 — Sept 2019		
4. TITLE AND SUBTITLE Triple Modular Redundancy MIPS Architecture Version 1.4				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 and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/EN/TR-19-03		
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Undisclosed				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 This report describes in detail the architecture of a Triple Modular Redundancy (TMR) MIPS processor based upon the Basic MIPS processor [1]. The TMR MIPS processor is used for Adaptive-Hybrid Redundancy research. There may be many other applications for the TMR MIPS processor beyond this specific research area.						
15. SUBJECT TERMS MIPS, Processor, Adaptive-Hybrid Redundancy						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maj Nicolas Hamilton, AFIT/ENG	
U	U	U	UU	114	19b. TELEPHONE NUMBER (include area code) (937) 255-6565 x4220; nicolas.hamilton@afit.edu	