MITIGATION OF RADIATION-INDUCED SOFT ERRORS USING TEMPORAL EMBEDDED SIGNATURE MONITORING

By

Daniel Limbrick

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

December, 2009

Nashville, Tennessee

Approved:

Professor William H. Robinson

Professor Bharat L. Bhuva

ACKNOWLEDGMENTS

First, I would also like to thank my advisor, Dr. William H. Robinson for his patience and insight both academically and personally. Dr. Robinson has helped me in every aspect of my graduate career from admission into the program to the completion of this Master's degree. Additionally, I want to thank Dr. Bharat L. Bhuva for his guidance and persistence in keeping me on track. I would also like to acknowledge Edward J. Ossi, Corey T. Toomey, and all members of the Radiation Effects and Reliability group that contributed insightful questions, comments, and suggestions to this work.

Finally, I would like to thank my parents, Donald and Patricia, for supporting me and providing me with the foundation to challenge myself academically. The path that they chose for themselves gave me the confidence to find my own. Without them, I would not be where I am today.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Soft errors are logical faults in a circuit's operation that do not reflect a permanent malfunction of the device. These errors are the result of particle strikes typically caused by: (1) alpha particles from package decay, (2) cosmic rays that produce energetic protons and neutrons, and (3) thermal neutrons [1],[2]. These particle strikes can be observed as flipped bits at the output of the affected node. If the strike occurs while the node is not in use (or not being latched), then the fault is masked from the output, and normal execution occurs. Because of masking, a circuit with a low frequency of soft errors could potentially be immune to a visible malfunction. However, there have been numerous studies to show that soft errors in microelectronics are a growing trend detected error to technology scaling [3-5].

When a soft error occurs in the control flow logic of a microprocessor, there is a risk that an incorrect instruction will be executed. This can cause incorrect data to be stored into memory or complete failure of the application currently executing. Because of this vulnerability, there has been significant work dedicated to solving this problem by monitoring the control flow of the program [6-8].

Control-flow monitoring techniques can be implemented at the hardware level, software level, or a combination of the two. Many control-flow error detection schemes use full software or hardware-assisted software techniques that involve redundant execution [7] or application-level watchdog timers [9]. However, these software

techniques usually rely on information from the application to implement error detection. The goal of this thesis was to implement a hardware-only control-flow error detection scheme, using the system state information that is only available within the microarchitecture and not visible at the application layer. In addition, the hardware used to monitor the control flow was minimized as a secondary goal.

This thesis presents a design to monitor the control flow of a processor by assigning a temporal signature to each instruction; the signature is based upon the remaining service time of the instruction. The processor considered as a testbed for this work was the MIPS R2000, which is a 32-bit processor implemented with five pipeline stages. Software-based fault injection simulations showed that the design detected over 80% of errors while running the Dhrystone synthetic computing benchmark. Logic synthesis results show that the monitoring circuitry increases the overall area of the MIPS processor by less than 1%.

The organization of this thesis is as follows. Chapter II presents a detailed explanation of the mechanisms associated with radiation-induced soft errors and control flow errors. Mitigation of radiation-induced soft errors implemented in the microarchitecture of a processor is described in Chapter III. Chapter IV gives a detailed description of the processor used in this study, including an overview of the instruction set architecture and the hardware description language (HDL) implementation. The TESM was first implemented using a Field-Programmable Gate Array (FPGA). The design was also synthesized for area, timing, and power analysis using a 45 nm CMOS technology cell library. Chapter V provides the full specifications and implementation of the design, referred to as a Temporal Embedded Signature Monitor (TESM). Chapter VI

describes the simulation and test setup for this project. Fault injection was conducted at critical nodes while running the Dhrystone benchmark. Finally, Chapters VII shows the results from the simulation and circuit synthesis and compares these results to the unmodified MIPS processor as well as other soft error mitigation techniques. Chapter VIII summarizes the work and describes future extensions of TESM.

CHAPTER II

IMPACT OF RADIATION-INDUCED SOFT ERRORS IN THE MICROARCHITECTURE

Overview of Microprocessor

Computers systems are designed collaboratively and modularly from many angles of perspective. In order to understand the reliability concerns of a microprocessor, these viewpoints (or abstraction levels) must be understood. A computer system can be separated into the following abstraction layers: Application, Middleware, Operating System, Instruction Set Architecture, Microarchitecture, Circuits, and Device Physics. Figure 1 shows the order of connectivity between these levels.

Applications are tools that function and are operated by means of a computer. Examples include word processors, spreadsheets, and media players. Applications are written in programming languages like C and Java. Middleware is the software that connects applications to the operating system. It generally consists of a library of functions that can allow applications to run without being specifically written for a particular operating system. Middleware is typically written in high level languages similar to those used for applications. An operating system coordinates tasks and manages hardware resources to optimize performance. Operating systems can be written in low-level programming languages like assembly, which is more closely mapped to the language that the hardware can interpret, or higher level languages like C. An Instruction Set Architecture (ISA) is the list and capabilities of all instructions that a processor can

execute as well as the specifications for the machine language. It acts as the interface between hardware and software. Microarchitecture is the description of the electrical circuitry of a computer necessary to implement the ISA. To implement the ISA and microarchitecture, Hardware Description Languages (HDLs) like Verilog and VHDL are used. The microarchitecture describes the logic gates used to implement the ISA. These logic gates can be created using circuits. Circuits are connections of components that are driven by current, such as resistors, capacitors, and inductors. An integrated circuit is a miniaturized circuit that has been fabricated on the surface of a thin substrate of semiconductor material. Microprocessors are an example of integrated circuits. Circuits can be designed and tested using schematic capture programs and simulators like Simulation Program with Integrated Circuit Emphasis (SPICE). Device physics are the mechanisms by which the circuit element is created including the materials used, the fabrication process, and the physical dimensions. This thesis focuses on improvements in the reliability of a microprocessor from the microarchitecture level.

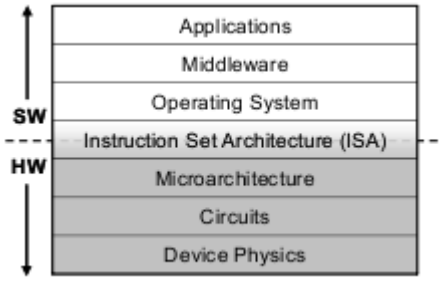

Figure 1: Levels of computer system abstraction [10]

Soft Errors

When an alpha particle or neutron strikes a circuit, it potentially generates charge

sufficient enough to cause a malfunction. At the device physics level, as seen in Figure 2, the particle can strike the drain of a transistor, interact with the molecular structure of the semiconductor material (usually silicon), and generate electron-hole pairs. These electron-hole pairs diffuse towards the device contacts. This diffusion creates current and interferes with the normal operation of the transistor. Additionally, the movement of charge carriers creates drift current that also disrupts normal operation. At the microarchitecture level, a particle strike at a logic gate's input node can cause an incorrect output to occur for as long as the additional charge remains on the node. If a particle strikes the input of a storage cell (i.e., latch), the incorrect output can be stored within that storage cell, provided that the strike occurs while the storage cell is accepting inputs.

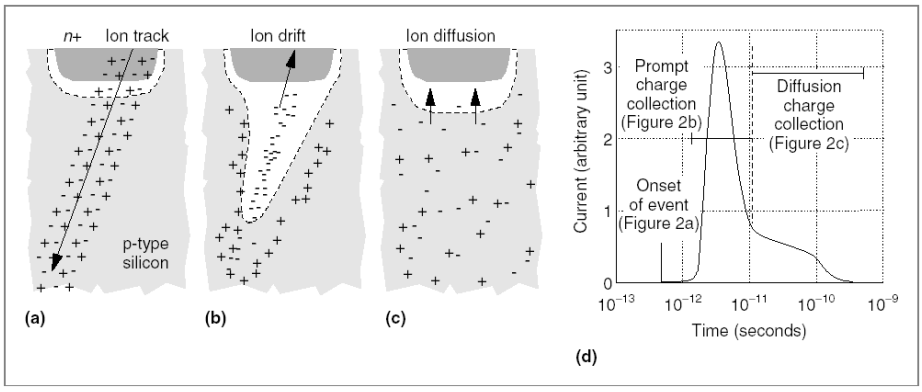

Figure 2: Charge generation and collection in a reverse-biased junction: (a) formation of a cylindrical track of electron-hole pairs, (b) funnel shape extending high field depletion region deeper into substrate, (c) diffusion beginning to dominate collection process, and (d) the resultant current pulse caused by the passage of a high-energy ion. [11]

The ability of a particle strike to induce an error that affects correct execution of a circuit can be measured in terms of the Mean-Time-To-Fail (MTTF) and the Failure-In-Time (FIT). The MTTF is a metric used to quantify the reliability of a circuit by observing the mean time expected for the first failure to occur. In order to determine the MTTF of two connected circuits that each have a known individual MTTF, Equation 1

can be used.

$$MTTF_{combined} = \frac{1}{\frac{1}{MTTF_1} + \frac{1}{MTTF_2}} \quad (1)$$

For easier calculation, the FIT metric is often used. One FIT means a failure occurs every billion hours. FIT relates to MTTF with Equation 2.

$$FIT = \frac{10^9}{24\,x\,365\,x\,MTTF \text{ in } years} \quad (2)$$

The FIT of a given circuit can be viewed as a measure of the masking properties of the circuit. First, an open logic path must exist through which the transient can propagate to arrive at a latch or other memory element. If the transient does not occur on such a path, then it is said to be logically masked. The amount of logical masking in a circuit is known as the Architectural Vulnerability Factor (AVF). Also, the transient must be of sufficient amplitude and duration to change the state of the latch or memory element. If the transient fails to meet this requirement, then it is considered electrically masked. This electrical characteristic is known as the Intrinsic FIT. Finally, in synchronous logic, the transient must arrive at a time when the clock pulse enables the memory element. Failure to meet this requirement means that the transient was latch-window masked. This characteristic is known as the Timing Vulnerability Factor (TVF). These factors are considered when calculating the FIT and can be seen in Equation 3.

$$FIT = \Sigma\,(AVF + TVF + Intrinsic\ FIT) \quad (3)$$

When planning the architecture of a microprocessor, the FIT value can be used as a design constraint for reliability. However, accounting for TVF and Intrinsic FIT is not possible at the architecture level. Instead, design decisions at the architecture level, which this thesis addresses, typically impact the AVF.

Architectural Vulnerability Factor

The physical manifestation of single events (e.g., transients, upsets) must occur in active computational structures to affect higher abstraction levels. Once a soft error is present, the impact on the software is dependent upon the architectural vulnerability factor as determined by the application executing on the IC. For soft-error reliability, architecture designers consider *undetected errors*, *true detected errors*, and *false detected errors*. This classification is similar to the error classification used by [12] and shown in Figure 3. If a soft error causes a bit flip but the bit is not used before it returns to a correct state, as seen in Outcome 1, then it is considered a benign fault. If the faulty bit is corrected, as seen in Outcome 2, then an error no longer exists. If this faulty bit is used, but does not affect the output of the program, as seen in Outcome 3, it is also considered benign. An example of this situation can be observed with an "OR" gate. Consider two input signals "A" and "B". If "A" is given the logic value "1", then the output of the "OR" gate will be "1" regardless of the value of "B". Therefore, a bit flip at "B" does not matter because it does not affect the outcome of the program. If the faulty bit matters and goes undetected, as seen in Outcome 4, then it is considered an undetected error. A benign error that is detected, as seen in Outcome 5, is considered a false detected error. A faulty bit that affects the output and is detected and not corrected, as seen in Outcome 6, is categorized as a true detected error.

Detected errors are potentially less dangerous to the operation of a microprocessor than undetected errors because they can be flagged by the hardware and mitigated by the application. For example, a memory structure can be monitored with parity and have a signal sent to the application when a parity mismatch occurs. Undetected errors do not

provide any information about the location or time of the error, leaving the system completely unprepared. Therefore, it is imperative to reduce the amount of undetected errors as much as possible.

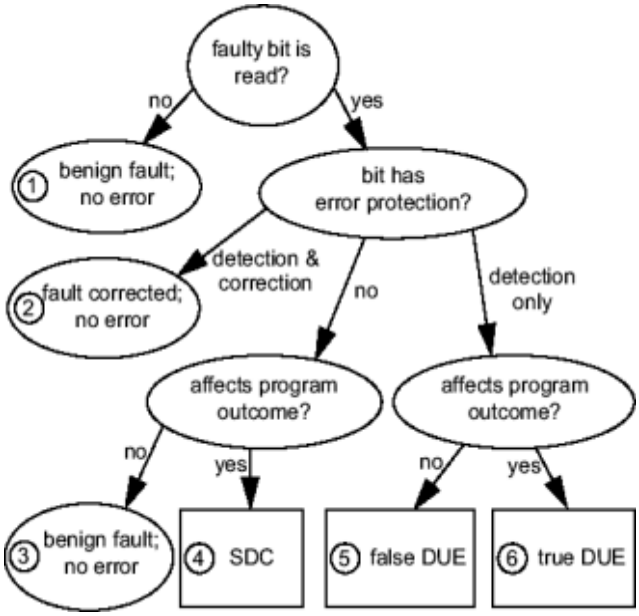

Figure 3: Soft Error Classification Flow Chart [12]. Silent Data Corruptions (SDC) are undetected bit-level errors. Detected Unrecoverable Errors (DUE) are detected bit-level errors that are not corrected.

Control-bit/Control-flow Errors

Control-bit errors have a significant impact on the program flow of a microprocessor. Control bits are the signals that activate the hardware necessary to execute the current instruction. For example, in the instruction `ADD R3, R1, R2`, the instruction code that signifies an **`ADD`** is found in the control bits. Additionally, the control bits inform the hardware that the value of two operands to be added can be found in Register 1 and Register 2, and the answer should be stored in Register 3. For this thesis, the control bits that determine which instruction will be executed are referred to as *control-flow bits*. When undetected errors occur in the control-flow bits of a

microprocessor, there is a risk that an incorrect instruction will be executed. This can result in incorrect data being stored into memory or complete failure of the application [12].

A previous investigation into the effects of errors on control-flow bits can be found in [13]. This study defined the following: (1) *operation errors* - a change in the operation code used, (2) *operand errors* - a change in or premature use of the register/operand addressed, (3) *execution errors* - a change in the functional units used, (4) *timing errors* - the instruction beginning or ending at an incorrect time, and (5) *order errors* - a commitment order violation. Through statistical fault injection simulations, it was determined that timing errors were the dominant group of control-flow errors. This result provided insight for the error detection approach discussed later in this thesis.

The goal of this thesis was to develop a method for reducing the impact of control flow errors in a microprocessor. In the 32-bit instruction word of a RISC processor, the control-flow bits (as defined above) are the "opcode" bits and the "funct" bits denoted in red in Figure 4.

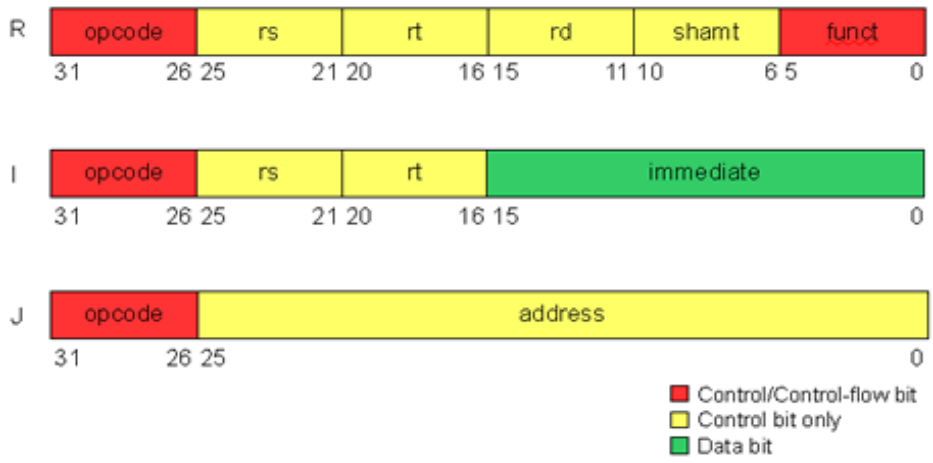

Figure 4: Distinction between control bits, control-flow bits, and data bits in a 32-bit implementation of the MIPS RISC processor.

CHAPTER III

MITIGATION OF SOFT ERRORS IN THE MICROARCHITECTURE

In order to protect against soft errors at the architecture level, several techniques have been used including error detection and correction (EDAC) codes [14], triple modular redundancy (TMR)[15], and built-in-self-test (BIST)[16]. EDAC works by generating additional bits that contain information about the data word and appending that to the data word. TMR requires three copies of a component operating simultaneously with their outputs compared and voted to eliminate single faults. This method can be seen in Figure 5. BIST allows for accurate soft error characterization which can be coupled with other mitigation techniques [17].

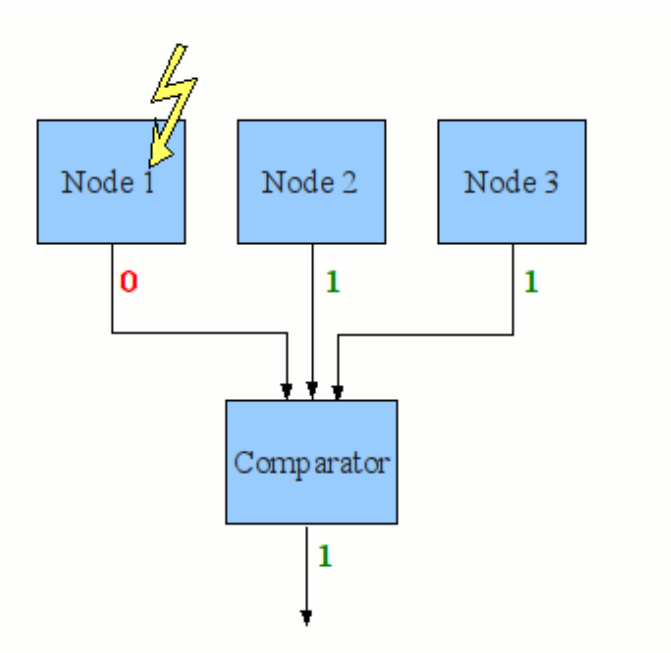

Figure 5: Example of Triple Modular Redundancy. A soft error occurs at Node 1 causing the bit to flip from "1" to "0". The other two nodes are unaffected. The three outputs are compared and the majority determines an overall output of "1".

Because of the critical vulnerability of control bits, there have been many solutions proposed to specifically solve this problem by monitoring the control flow of the program [6-8], including the solution in this thesis, embedded signature monitoring.

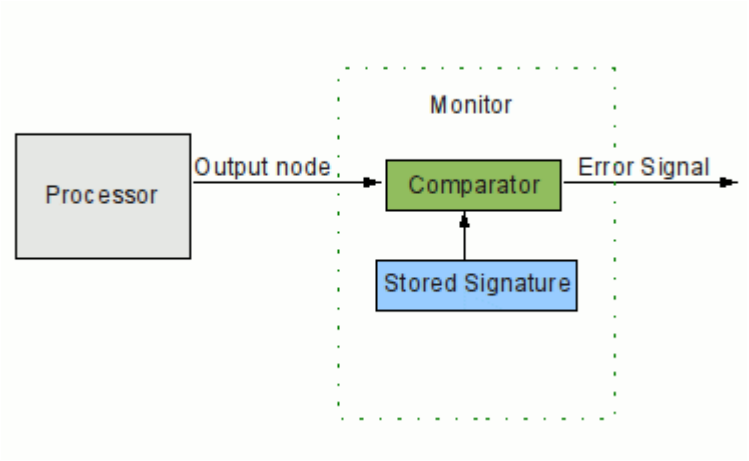

Figure 6: A General Signature Monitor

Embedded signature monitoring is used to check the control flow of a microprocessor. Usually, the monitor receives an instruction and interprets a pattern specific to that instruction, known as a signature. This signature is a numeric symbol that can represent any known behavior about the output node. For instance, in [18], the signature is the ordered list of instructions to be executed by the program, as determined during compilation. The monitor stores the signature (either statically hard-coded or dynamically obtained during run-time) at the beginning of execution and compares it with the information obtained in a later execution step. A visual representation of this setup can be seen in Figure 6. The signature is usually generated using code compaction hardware like linear feedback shift registers. This allows for the signature generating hardware to be small relative to complete duplication of the hardware. The design of the embedded signature monitor used for this study is discussed in Chapter V.

CHAPTER IV

MIPS PROCESSOR

The MIPS R2000 processor from [19] was used as a testbed to implement the control-flow monitor. A MIPS processor is a Reduced Instruction Set Computer (RISC). The MIPS R2000 structure is shown in Figure 7. The processor completes an instruction in five stages: Fetch, Decode, Execute, Memory, and Write-Back. In the Fetch stage, the instruction is loaded from memory based on the address given by the program counter. In the Decode stage, the instruction word is separated into the control bits necessary to execute the instruction and the operands that will be used by the instruction. In the Execute stage, the operation specified by the instruction is executed. In the Memory stage, any calls to memory that are necessary for instruction completion are performed. In the Write-back stage, the instruction writes its result into the register file.

The MIPS R2000 processor has a 32-bit word length, and instructions have three formats: R-type, I-type, and J-type. Figure 4 gives a visual representation of for the instruction word separated into each format. R-type instructions are typically for instructions that require three operands. I-type instructions are for instructions like load, store, or branch that use immediate constants. J-type instructions are for jump instructions that significantly alter the program counter.

The control-flow bits of the 32-bit instruction word are the opcode (instruction word bits 31-26) and the function code (instruction word bits 5-0). The control-flow monitor aims to protect these 12 bits. However, each instruction does not require all of

the control bits. For R-type instructions, the opcode field (bits 31 to 26) and the function code (bits 5 to 0) are required for correct control flow. For I-format and J-format instructions, only the opcode (bits 31 to 26) are required for correct control flow.

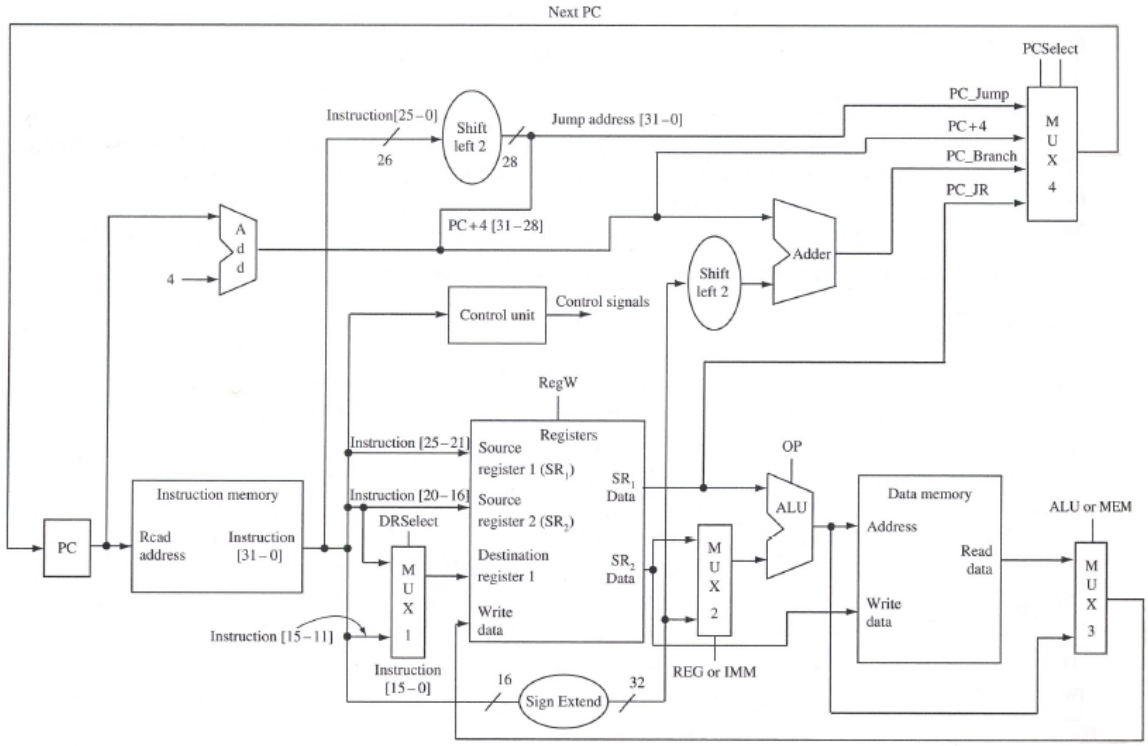

Figure 7: MIPS R2000 Architecture [19]

The cycle-by-cycle instruction flow for the MIPS R2000 processor can be seen in Figure 8. Unconditional branches using an immediate operand require two cycles to complete. Unconditional branches using a register and conditional branches require three cycles to complete. Arithmetic, logical, and store operations require four cycles to complete, and load operations require five cycles to complete. Accounting for single-bit flips only, each instruction has a limited number of soft error bit-flip combinations that will transform it into another realizable instruction. For instance, the load instruction is assigned the operation code “100011” and takes 5 cycles to complete. If a bit-flip occurs

on the most significant bit, the operation code will be transformed to “000011”, which does not match the operation code for any other instruction in the instruction set. The time for this unknown instruction to complete depends upon the cycle in which the fault was injected (and typically occurs in that cycle). However, if a bit-flip occurs on the third most significant bit, the operation code will be transformed from “100011” to “101011”, which is equivalent to the operation code for the store instruction. This could cause the instruction to finish in 4 cycles. From this knowledge, an inherent susceptibility to soft errors and detectability of the TESM can be predicted. In the general case, each instruction is susceptible to a soft error at least for bit flips that cause a transition to another realizable instruction. This can be considered the lower bound for error susceptibility since it assumes that a change to an unrealizable instruction finishes in a time different from the original instruction. For the load instruction, only one of the six opcode bits can cause a transition to another realizable instruction (the store instruction) so the error susceptibility is 16.67%. Therefore, arithmetic, logical, and store operations that encounter a soft error will only generate a unique signature if the bit-flip causes the instruction to be read as a branch instruction or a load instruction.

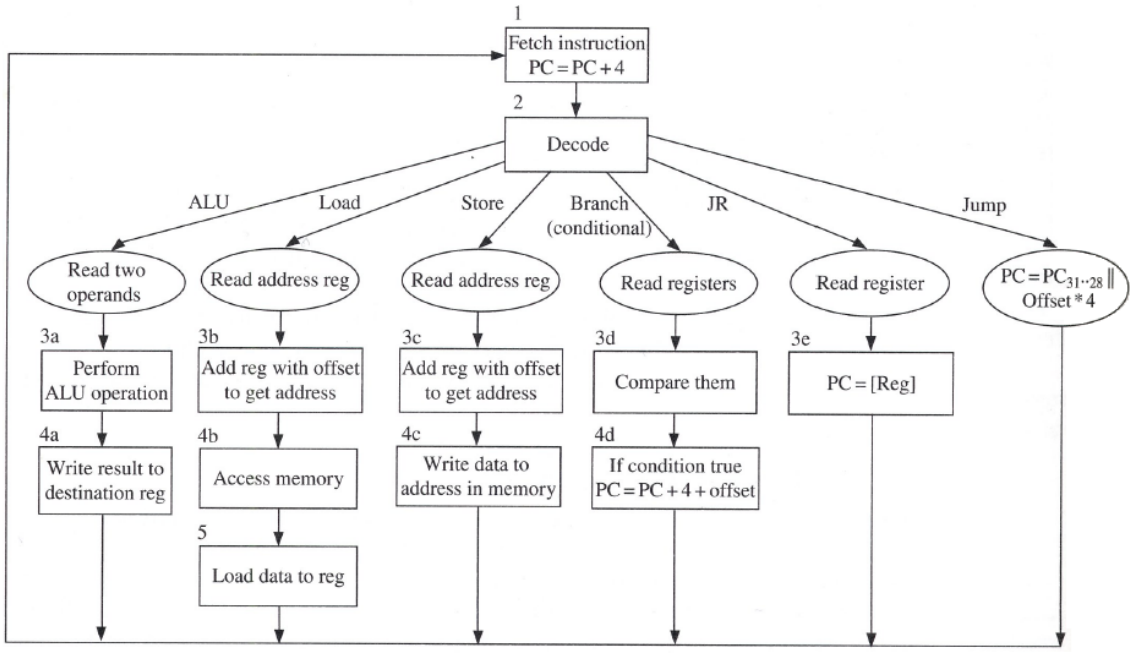

Figure 8: MIPS Instruction Flow Chart [19]

Instruction	Opcode	Single-bit transitions	Cycle Time	% Susceptible	% Detectable
Load	100011	Store	5	16.67	100
Store	101011	Load	4	16.67	100
ALU	000000	J, BNE, ADDI, JR*	4	66.67	75
Add Immediate	001000	ALU/JR, ANDI, ADDIU	4	50	25
Add Immediate Unsigned	001001	ORI, ADDI	4	33.33	0
And Immediate	001100	ORI, ANDI, BEQ	4	50	33.33
Or Immediate	001101	ANDI, BNE	4	33.33	50
Jump	000010	ALU/JR	2	16.67	100
Branch on Equal	000100	ALU/JR, BNE, ANDI	3	50	50
Branch on Not Equal	000101	BEQ, ORI	3	33.33	50
Jump Register	000000	J, BNE, ADDI, ALU*	3	66.67	75

* - transition caused by error in funct bits

Table 1: Error susceptibility of each instruction by mathematical reasoning

VHDL Implementation

In the design of a microprocessor, a formal description of the digital logic is written using a Hardware Description Language (HDL). HDLs allow circuit designer to represent hardware semantics without mapping the design to a specific technology. For instance, when a full adder is specified in an HDL, it is known that the gate receives two inputs and generates an output equivalent to the addition of the inputs. However, it is not known how the full adder will be implemented when the circuit is fabricated. A full adder can be implemented using various combinations of gates. For instance, a full adder can be implemented using only NAND gates or by using a combination of XOR and Logical AND gates. Also, the logic that a full adder represents can be built using CMOS transistors, NMOS transistors, bipolar junction transistors, etc. In addition, the transistors used to build the gate are available in various sizes, causing performance characteristics to vary as well.

An advantage of using an HDL is that a circuit designer can ensure that the logic of the circuit design operates correctly before committing the resources to build the circuit. This pre-build testing can be accomplished by using a HDL Integrated Development Environment (IDE). HDL IDEs can contain a source code editor, a compiler that parses the HDL code to determine syntactic correctness, a simulator that interprets the behavior of the code as if it were implemented with hardware, and a debugger.

The description for the MIPS R2000 processor used in this thesis was written in the Very-High-Speed Integrated Circuits Hardware Description Language, or VHDL. The VHDL IDE used for this thesis was Altera's Quartus II software in combination with

ModelSim.

CHAPTER V

TEMPORAL EMBEDDED SIGNATURE MONITOR

Control flow monitoring techniques can be implemented at the hardware level, software level, or a combination of the two. The design used in this thesis implements a hardware-only control flow error detection scheme. This design satisfied the goal of exposing system information that is not available to the application, thus converting an undetected error to a detected error. In addition, the hardware used to monitor the control flow is minimized.

Design Description

The control-flow monitor receives 12 control bits (specific to the MIPS R2000 processor) during the same cycle that the decode stage receives the instruction in the processor. The monitor decodes the control bits into the amount of time (in cycles) required to complete the given instruction. This information is processor specific. This decoded information is sent through a register file that is synchronized with the processor's instruction pipeline. When an instruction commits, a signal is sent to the register that contains the timing information. For instance, in the MIPS R2000 architecture, an **`ADD`** instruction takes three cycles after the fetch stage (four cycles in total) to complete. When the **`ADD`** instruction completes, it will send a signal to the control-flow monitor and the monitor will look in the third register to ensure that it contains the signal indicating a four-cycle instruction completion. This method takes

advantage of temporal and spatial redundancy because the information has to be in the correct register, and it has to contain the correct time. It also simplifies the design because additional hardware is not necessary to catch the correct instruction codes on the processor bus during times that the processor is not fetching an instruction. A visual representation of this method can be seen in Figure 9. Figure 10 provides a gate-level view.

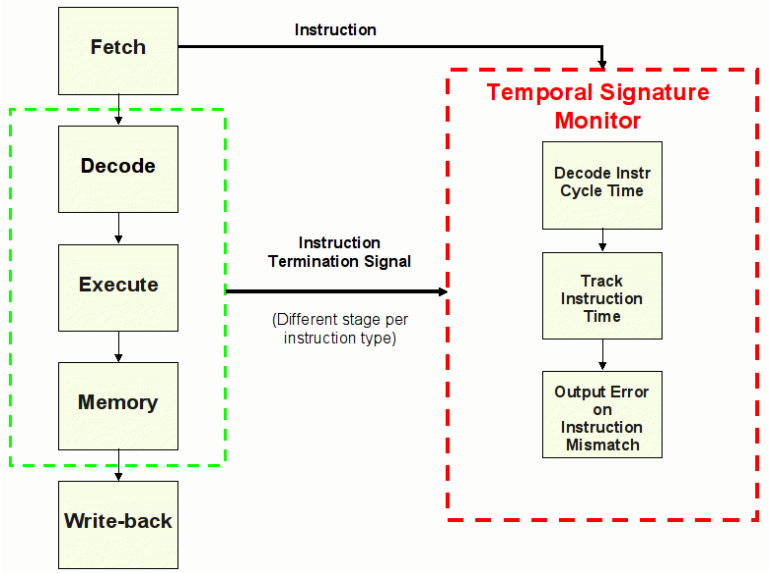

Figure 9: Flow Chart of TESM operation

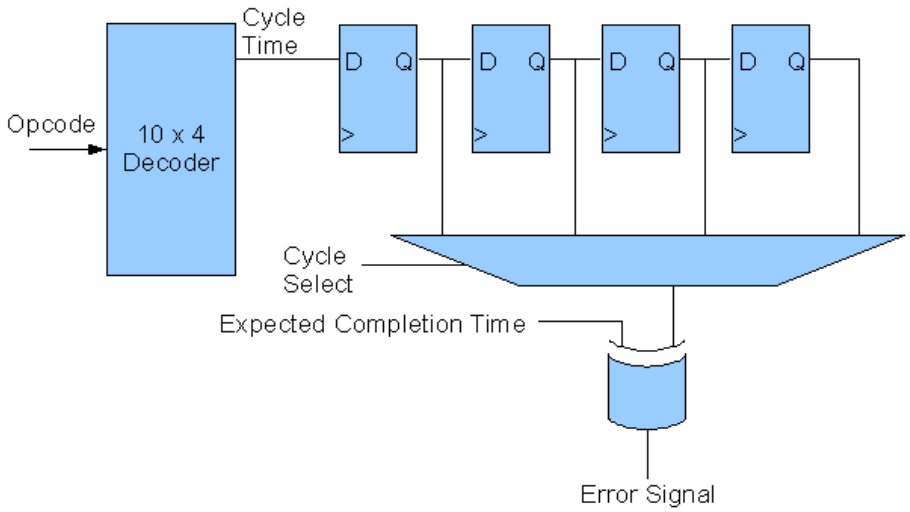

Figure 10: Gate level view of the temporal embedded signature monitor

Comparison to Related Designs

A similar idea to this is incorporated in [8] where a shadow register file was used to verify the contents of the registers for instructions that the application deemed critical. In this design, the register file contains the number of cycles for each instruction. The comparison that occurs is between the amount of time it took for the instruction to execute and the amount of time that the monitor decoded in the decode cycle of the instruction. This design has the advantage of completely removing detection responsibility from the application level.

This design increases in size minimally with increased instruction complexity. Consider *n* as the number of distinct possible times it takes for a processor to complete an instruction. The register file that holds the completion time information would need $\log_2$ (n) registers. For instance, the MIPS R2000 processor used for this thesis is capable of executing 31 types of instructions with four possible distinct execution times. It requires two registers to represent the four possible values for completion time.

CHAPTER VI

EXPERIMENTAL METHODOLOGY

The VHDL model that was created for our design was implemented in an Altera DE2 Development and Education Board which uses a Cyclone II Field Programmable Gate Array (FPGA). To simulate a typical workload, the Dhrystone benchmark was chosen as the application that our design would run. A software-based fault injection method was used to simulate soft errors and determine the effect of these errors on the circuit.

Instruction Characterization

In order to test the error detectability of an instruction, a simulation was performed on each instruction. In this simulation, each instruction was tested with known inputs and outputs, and a fault was injected during execution. The output of that instruction was compared to the expected output to see if the error was detected correctly.

Dhrystone Benchmark

The Dhrystone benchmark is a synthetic computing benchmark developed by Reinhold P. Weicker. Developed in 1984, it was one of the first industry standard benchmarks to represent general CPU performance for integer operations. The C version of this benchmark, used in this study, was created by Rick Richardson. The performance metric for the Dhrystone benchmark is the number of iterations of the main loop code per

second, known as a “Dhrystone MIPS”.

The Dhrystone benchmark was used for this thesis for many reasons. The benchmark is an indicator of general-purpose performance of computers and has remained in broad use in the embedded computing world [20]. Also, the Dhrystone benchmark gives a representative distribution of instructions that the MIPS R2000 is capable of executing; this characteristic is important for fault-injection analysis. Additionally, the Dhrystone benchmark has a relatively small number of instructions, making the simulation time more practical. A similar test setup for this processor can be found in [21].

The Dhrystone benchmark is composed of 8 main “procedures” and 3 main “functions”. In this thesis, the instructions of the main procedures and functions were used with fault injection to test the effectiveness of the TESM. The frequency of instructions that each function and procedure contains can be seen in Table 2.

	ANDI	ADDI	ADDIU	ORI	J	BEQ	BNE	ALU	SW
Proc 1	0	0	14	0	0	1	0	5	32
Proc 2	0	0	2	0	0	0	0	2	1
Proc 3	0	0	1	0	1	0	0	0	1
Proc 4	0	0	0	0	0	0	0	2	1
Proc 5	0	0	0	0	0	0	0	1	1
Proc 6	0	0	4	0	0	2	0	2	8
Proc 7	0	0	1	0	0	0	0	1	1
Proc 8	0	0	4	0	0	0	0	0	8
Func 1	2	0	0	0	0	1	0	2	0
Func 2	1	0	9	0	0	1	0	1	8
Func 3	0	0	0	0	0	0	0	1	0

Table 2: Instruction distribution for each section of the Dhrystone benchmark

From the table, it can be observed that “Proc 1”, “Proc 6”, and “Func 2” have the largest

number of instructions and therefore provide the most information. Also, all sections of the benchmark are dominated by load and store operations. Based on the high percentage of memory operations, the Dhrystone benchmark results should resemble the frequency of results from the individual load and store tests.

Fault Injection Procedure

Fault injection is a technique used to test the reliability of a circuit by introducing faults in locations of interest and observing the effect they have on the output of the circuit. Fault injection mechanisms can be classified into two areas: hardware-based and software-based. Hardware-based fault injection involves using equipment to physically mimic an SET. An example of hardware-based fault injection with direct hardware contact is using a power supply to apply a voltage to a test point. Hardware-based fault injection with indirect contact is performed with a laser beam, proton accelerator, or any other device that can mimic an SET without applying a probe. Software-based fault injection involves using a stimulus in the programming environment to invert a bit value. This can be done either at run-time or during compile-time. For compile-time testing, the program instruction is modified before the image is loaded and executed. For run-time testing, the fault injection is triggered by a mechanism like time-out, exception, or code-insertion [22].

Hardware-based fault injection is beneficial because the user has good controllability of the fault injection times. Also, there is little to no perturbation of the target system. In other words, the target system is almost identical to the system that will be used. An additional advantage is that hardware-based fault injection mimics the

natural physical phenomena of fault injection and therefore gives a relatively accurate depiction of how a system will react in a natural radiative environment.

A major disadvantage to using this fault injection mechanism is that it is costly. Once a system has been tested using a proton or heavy ion beam, the permanent radiation damage prevents the target system from being used in practice and for future testing. Another key issue is accessibility to a hardware-based fault injection testing environment. Currently there are fewer than 30 cyclotrons in the United States. In order to conduct a test, it is necessary to schedule a test session and travel to one of these locations. This can lengthen the time it takes to verify that a system is radiation-hardened.

Software-based fault injection is beneficial because it does not require expensive hardware. Simulations can be done with no cost by inserting additional fault-injection code into the VHDL model. An additional advantage is that software-based fault injection can target specific applications and operating systems. This speeds up the test time because the user does not have to wait for critical errors.

Software-based fault injection is the method used in this thesis to determine the architectural vulnerability of the MIPS R2000 processor to soft errors and the effectiveness of the TESM.

Methodology

Soft errors were simulated by using a VHDL description of XOR gates with the control bits and a 12-bit fault injection signature as the inputs, shown in Figure 11. This method is similar to the one proposed in [23]. Faults were injected one at a time into the control bits of every possible instruction during each program flow cycle. A high-level view of the fault injection locations can be seen in Figure 12.

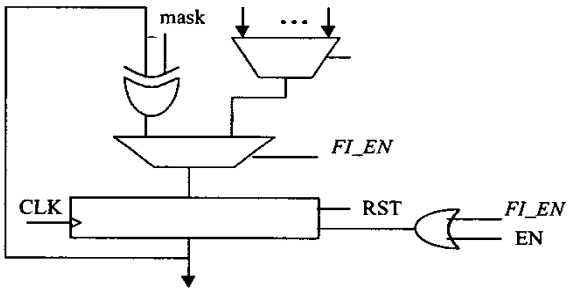

Figure 11: XOR gate fault injection model [23]

For fault injection, two test-setups were used. In the first test setup, two copies of the MIPS processor were instantiated. The first copy contained the original MIPS VHDL code and can be considered the "golden" copy. The second copy contained nodes with fault injection capability and can be considered the "dirty" copy. These copies were instantiated in an outer module that acted as a test logger. The Dhrystone benchmark was run on both copies and faults were injected into the "dirty" copy. At the end of each trial the results of each instruction were recorded by the test logger and compared to the original results data. This data was used to quantify the inherent vulnerability of the MIPS R2000 processor.

The second test setup was similar to the first setup except that the "dirty" copy was replaced with the TESM-modified MIPS processor. The same procedure was run, and the test logger recorded the differences in output. This data was used to determine the effectiveness of the TESM in detecting timing errors.

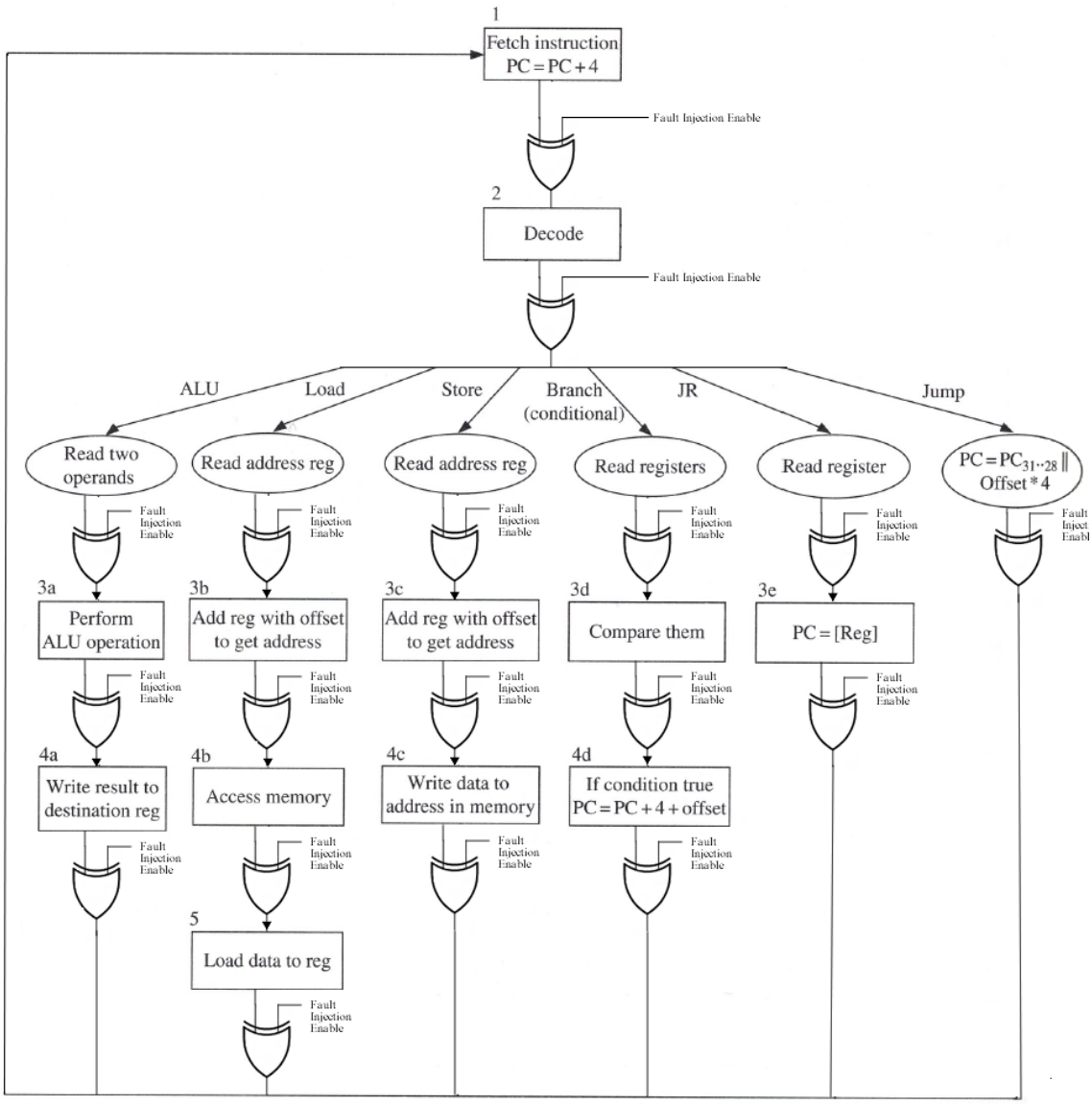

Figure 12: High-level view of the fault injection locations used for testing

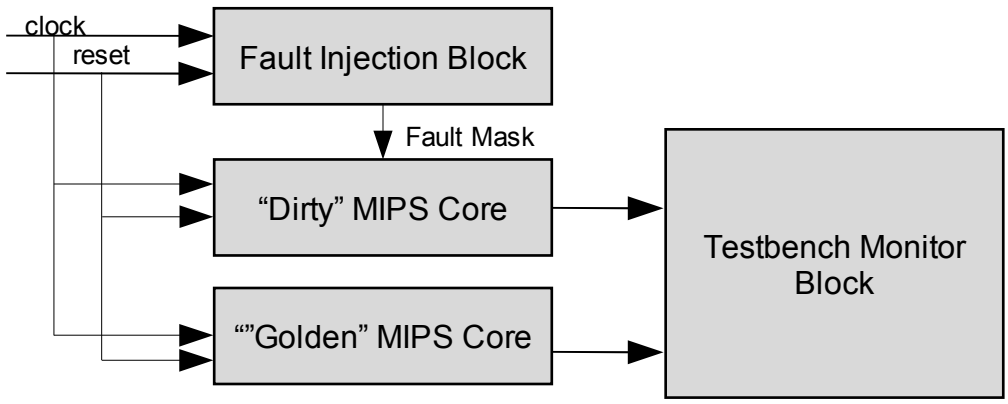

Figure 13: Block diagram of fault injection test setup

The monitor block shown in Figure 13 is a VHDL testbench run in ModelSim version 6.1. It contained the following concurrently running processes. The “Initial” process contains the initial parameters for the testbench including the triggers to reset the test. The “Clock” process sets the clock period to 100 nanoseconds. The “Run-time” process manually records the run time of the simulation excluding the load time. The “Loading” process loads the Dhrystone benchmark into the instruction memory of the MIPS processor. The “Error Detection” process checks the error flag once per rising clock edge. Once an error is detected the process waits for the simulation is reset. The “Fault Generation” process activates one of the 35 possible fault injection nodes. The process rotates the fault location once per test. The “Fault Propagation” process checks for the “commit” signal from the “golden” copy on each rising clock edge. When the “commit” signal of the “golden” copy is asserted, the process checks the “commit” signal, data bus, and memory bus of “dirty” copy.

Logic Synthesis

In the microprocessor design process, after the VHDL description of the circuit is

written and tested, it is mapped to a specific technology. This mapping from an HDL description to logic gates in a technology cell library is referred to as logic synthesis. Synthesizing the circuit provides the fabrication layout for the design. At this level, the physical characteristics of the circuit such as the area, maximum clock speed, and power consumption can be obtained based on the technology used.

The MIPS R2000 processor and our design were synthesized to the FreePDK45[24] cell library using Cadence RTL Compiler for power, timing, and area information. The FreePDK45 cell library was developed by the Oklahoma State University VLSI Computer Architecture Group; it consists of 33 cells with a 45-nm transistor size. This library was developed based on the official scalable CMOS (SCMOS) design rules of the Metal Oxide Semiconductor Implementation Service (MOSIS). MOSIS is one of the oldest semiconductor fabrication plants. The FreePDK45 library was chosen because it was an open-source implementation of a current fabrication technology.

CHAPTER VII

RESULTS AND DISCUSSION

The results for the instruction characterization can be seen in Figure 14. The TESM can detect approximately 60% of all control-flow errors that are observable on the output of the microprocessor.

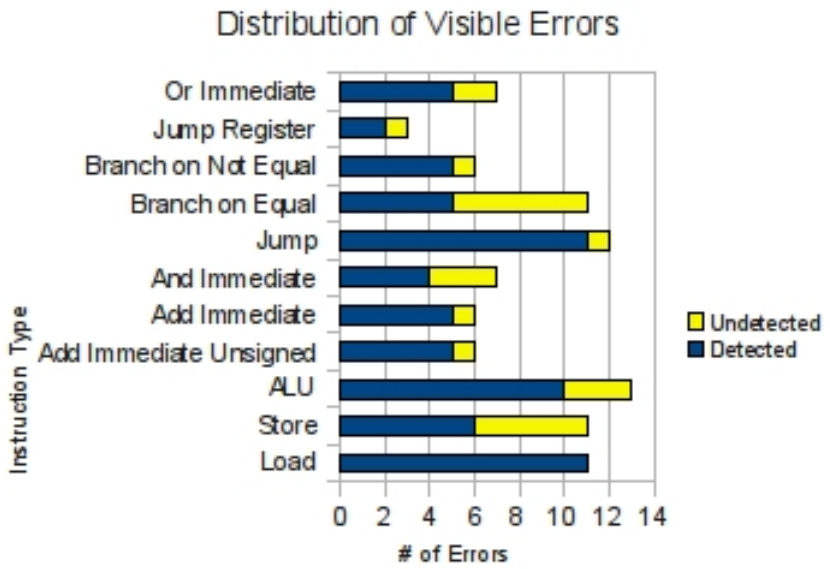

Figure 14: Distribution of detected errors

The results for the fault injection simulation with the Dhrystone benchmark can be seen in Figure 15. The results show that the TESM detected an error 81% of the time. In both tests, no false detections occurred. Figure 16 shows the distribution of errors separated by sections of the Dhrystone benchmark.

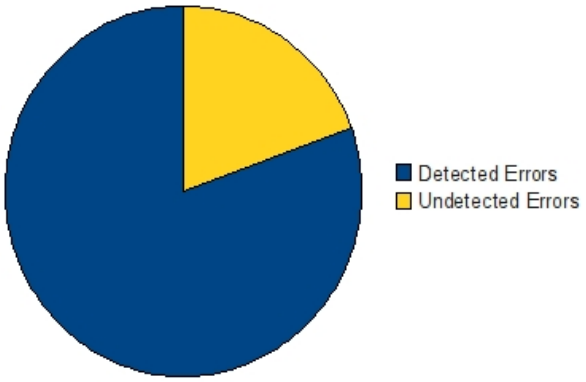

Figure 15: Error distribution of Dhrystone benchmark

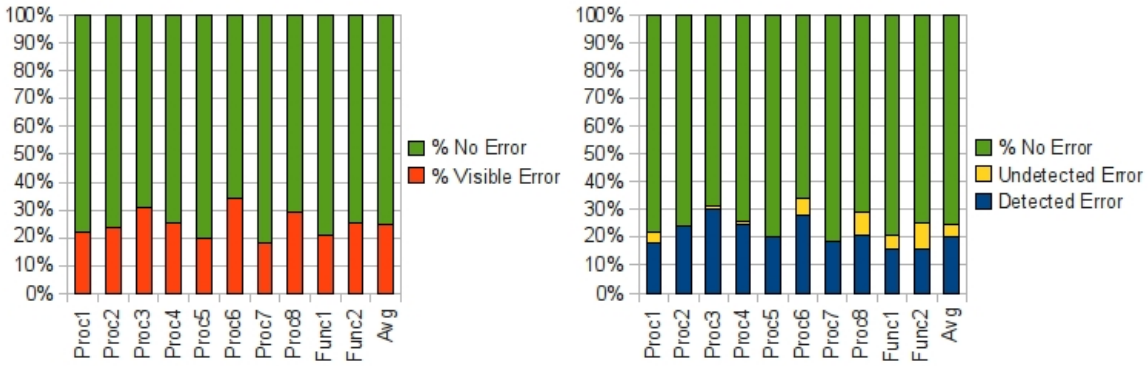

Figure 16: Error by type distribution for each section of the benchmark

The power, timing, and area information for the MIPS R2000 processor and our hardened MIPS R2000 processor design can be seen in Table 3. As the table shows, the addition of our control flow monitor has minimal effects on the maximum clock frequency and area of the circuit with a slightly greater effect on power consumption.

Table 3: Comparison of area, timing, and power between the original MIPS and the MIPS with the temporal embedded signature monitor

	Area (mm^2)	Timing (MHz)	Power (mW)
Original MIPS	33,414	222	1.44
MIPS w/ TESM	33,665	216	1.61
% increase	0.75	-2.64	11.73

The difference in error detection rate for the single instruction testing compared to the Dhrystone benchmark testing could be detected error to the high volume of memory instructions executed in the Dhrystone benchmark. As seen in Figure 16, the sections of the benchmark with the most undetected errors, “Proc 6”, “Proc 8”, and “Func 2”, also contain the largest percentage of ADDIU instructions. This trend is consistent with the inherent vulnerability of the ADDIU instruction.

The results show that using the temporal embedded signature monitor design with the MIPS processor improves the reliability of the processor with less than 1% increase in area. This is a significant reduction for such a small area penalty. Consider a parity check circuit for the control flow bits of each stage in the processor. With 12 potential control-flow bits per stage, and a 5-stage pipeline, a simple parity checker implementation would require approximately 60 XOR gates. Synthesizing this implementation with the same library would potentially result in twice the increase in area compared to the TESM. Similarly, dual modular redundancy implemented on the control-flow bits would require approximately 60 XOR gates and would result in a similar area increase. In both situations, the number of false detected errors would increase significantly because errors will be detected that do not propagate to the commit stage. Additionally, for processors with longer pipelines, the TESM should still maintain a relatively small area increase because it only calls for two additional flip-flops per pipeline stage. The parity and dual modular redundancy implementations would require 12 additional gates per stage.

According to the results, the TESM was most effective for non-R-type instructions. This is consistent with predictions because R-type instructions in the MIPS processor complete in the same number of cycles. Therefore errors in R-type instructions

that convert the instruction to other R-type instructions will go undetected. Memory instructions were predicted to have 100% coverage because they have unique completion times from any other instruction. The Dhrystone benchmark results indicate that this prediction is accurate.

Note that using the TESM does not preclude the use of other detection mechanisms. With less than 1% area overhead, the TESM could be joined with parity or dual modular redundancy on the ALU to provide more detection but with less area penalty than a full parity or dual modular redundancy implementation. The substantial increase in power can be attributed to the TESM operating during all stages (Fetch, Decode, Execute, Memory, Write-back) of the instruction flow. However, the switching activity of the MIPS processor is separated by stage. For instance, the Fetch, Decode, Execute, and Memory stages are static while the instruction is in the Write-back stage. In a pipelined implementation, the switching power of the MIPS processor would be greater because each stage would contain constantly switching signals. Therefore, the TESM would contribute to a smaller percentage of the overall power.

CHAPTER IX

CONCLUSION

This thesis presented a hardware technique to detect errors in the control flow of a MIPS R2000 processor. This design can be used as a low-cost measure to reduce control-flow soft errors in microprocessors. An expansion of this study could include implementing the TESM on a larger instruction set to compare the area increase with instruction set size. Also, testing the TESM with a pipelined MIPS implementation should produce similar results presented in this thesis so including this testing in a future study could provide verification. Additionally, the TESM can be expanded to work with instruction sets of greater complexity and coupled with techniques to mitigate the soft errors once they have been detected.

APPENDIX A

TESM VHDL BEHAVIORAL DESCRIPTION

This appendix displays the VHDL behavioral description of the temporal embedded signature monitor discussed in this thesis.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;

entity TESM is
  port(     CLK:        in std_logic;
            Instr:      in UNSIGNED(31 DOWNTO 0);
            Read_In:    in STD_LOGIC_VECTOR(1 DOWNTO 0);
            Check_In:   in STD_LOGIC;
            Err_Flag:   out STD_LOGIC_VECTOR(1 DOWNTO 0)
      );
end TESM;

architecture Behavioral of TESM is
  SIGNAL cnt: STD_LOGIC_VECTOR(1 DOWNTO 0);
  SIGNAL cnt2: STD_LOGIC_VECTOR(1 DOWNTO 0);
  SIGNAL stage1: STD_LOGIC_VECTOR(1 DOWNTO 0);
  SIGNAL stage2: STD_LOGIC_VECTOR(1 DOWNTO 0);
  SIGNAL stage3: STD_LOGIC_VECTOR(1 DOWNTO 0);
  constant lw:   UNSIGNED(5 downto 0) := "100011";  -- 35
  constant jump: UNSIGNED(5 downto 0) := "000010";  -- 2
  constant jump_register: UNSIGNED(5 downto 0) := "001000"; -- 8
  constant beq:  UNSIGNED(5 downto 0) := "000100";  -- 4
  constant bne:  UNSIGNED(5 downto 0) := "000101";  -- 5
  constant dont_care: UNSIGNED(5 downto 0) := "000000";
  constant r_type: UNSIGNED(5 downto 0) := "000000";

  alias opcode: UNSIGNED(5 downto 0) is Instr(31 downto 26);
  alias F_Code: UNSIGNED(5 downto 0) is Instr(5 downto 0);

begin

-- DECODER
      WITH opcode SELECT
            cnt <="00" WHEN jump,   -- Jump takes 2 cycles
                  "11" WHEN lw,     -- Load takes 5 cycles
                  "01" WHEN beq,    -- Conditional branch takes 4 cycles
                  "01" WHEN bne,    -- Conditional branch takes 4 cycles
                  "10" WHEN OTHERS; -- R_type or Jump Register

      WITH F_Code & opcode SELECT
            cnt2 <= "01" WHEN "001000" & r_type,     -- Jump Reg takes
```

```
3 cycles
                cnt WHEN OTHERS;  -- other R_type/I_type instructions
take 4 cycles
-- END DECODER

      WITH Read_In & Check_In SELECT
            Err_Flag <= cnt2 XOR Read_In  WHEN "001",
                        stage1 XOR Read_In  WHEN "011",
                        stage2 XOR Read_In  WHEN "101",
                        stage3 XOR Read_In  WHEN "111",
                        "00"  WHEN OTHERS;

      regs: PROCESS (clk)
      BEGIN
            if rising_edge(clk) then
                  stage1 <= cnt2;
                  stage2 <= stage1;
                  stage3 <= stage2;
            end if;
      END PROCESS regs;
end Behavioral;
```

APPENDIX B

HARDENED MIPS WITH FAULT INJECTION VHDL BEHAVIORAL DESCRIPTION

This appendix displays the VHDL behavioral description of the MIPS R2000 processor with the fault injection nodes and the temporal embedded signature monitor that was discussed in this thesis.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity MIPS is
  port(CLK, RST: in std_logic;
       CS, WE, IRAM_select: out std_logic;
         ERR_MASK: in UNSIGNED(35 DOWNTO 0);
       ADDR: out unsigned (31 downto 0);
       Err_Flag:  out STD_LOGIC_VECTOR(1 DOWNTO 0);
         Commit: out std_logic;
         nState_OUT: out std_logic;
      WD_Check: out std_logic;
       Mem_Bus: inout unsigned(31 downto 0));
end MIPS;

architecture structure of MIPS is
  component REG is
    port(CLK: in std_logic;
         RegW: in std_logic;
         DR, SR1, SR2: in unsigned(4 downto 0);
         Reg_In: in unsigned(31 downto 0);
         ReadReg1, ReadReg2: out unsigned(31 downto 0)
      );
  end component;

  component Watchdog_Timer is
      port(CLK:         in std_logic;
       Instr:           in UNSIGNED(31 DOWNTO 0);
       Read_In:   in STD_LOGIC_VECTOR(1 DOWNTO 0);
       Check_In:  in STD_LOGIC;
       Err_Flag:  out STD_LOGIC_VECTOR(1 DOWNTO 0)
      );
      end component;

--  SIGNAL timer_cnt: unsigned := '1';
--  SIGNAL test1, test2, test3, test4: unsigned := '0';

  type Operation is (and1,or1,add,sub,slt,shr,shl,jr,add2);
  signal Op, OpSave: Operation := and1;
```

```
  type Instr_Format is (R, I, J);  -- (Arithmetic, Addr_Imm, Jump)
  signal Format: Instr_Format := R;
  signal Instr, Imm_Ext: unsigned (31 downto 0);
  signal PC, nPC, ReadReg1, ReadReg2, Reg_In: unsigned(31 downto 0);
  signal ALU_InA, ALU_InB, ALU_Result: unsigned(31 downto 0);
  signal ALU_Result_Save: unsigned(31 downto 0);
  signal ALUorMEM, RegW, FetchDorI, Writing, REGorIMM: std_logic :=
'0';
  signal REGorIMM_Save, ALUorMEM_Save: std_logic := '0';
  signal DR: unsigned(4 downto 0);
--  signal State: integer range 0 to 4 := 0;
  signal State, nState : integer range 0 to 4 := 0;

  signal WD_Read_In: STD_LOGIC_VECTOR(1 DOWNTO 0) := "00";
  signal WD_Check_In: STD_LOGIC;
  --signal WD_Check_In_OUT: STD_LOGIC;

  signal F_Code2: unsigned (5 downto 0);       -- Added for Error
Injection
  signal Opcode_State0: unsigned (5 downto 0);       -- Added for Error
Injection
  signal Opcode_State1: unsigned (5 downto 0);       -- Added for Error
Injection
  signal Opcode_State2: unsigned (5 downto 0);       -- Added for Error
Injection
  signal Opcode_State3: unsigned (5 downto 0);       -- Added for Error
Injection
  signal Opcode_State4: unsigned (5 downto 0);       -- Added for Error
Injection

  constant addi: unsigned(5 downto 0) := "001000";  -- 8
  constant andi: unsigned(5 downto 0) := "001100";  -- 12
  constant ori:  unsigned(5 downto 0) := "001101";  -- 13
  constant lw:   unsigned(5 downto 0) := "100011";  -- 35
  constant sw:   unsigned(5 downto 0) := "101011";  -- 43
  constant beq:  unsigned(5 downto 0) := "000100";  -- 4
  constant bne:  unsigned(5 downto 0) := "000101";  -- 5
  constant jump: unsigned(5 downto 0) := "000010";  -- 2
-- Added Instructions --
  constant addiu: unsigned(5 downto 0) := "001001";  -- 9  WORKS!
  alias opcode: unsigned(5 downto 0) is Instr(31 downto 26);
  alias SR1: unsigned(4 downto 0) is Instr(25 downto 21);
  alias SR2: unsigned(4 downto 0) is Instr(20 downto 16);
  alias F_Code: unsigned(5 downto 0) is Instr(5 downto 0);
  alias NumShift: unsigned(4 downto 0) is Instr(10 downto 6);
  alias ImmField: unsigned (15 downto 0) is Instr(15 downto 0);
begin

  WD_CHECK_IN <= '1' WHEN nState = 0 else '0';
  WD_CHECK <= WD_CHECK_IN;

  WD1: TESM port map(
            CLK => CLK,
            Instr => Instr,
            Read_In => WD_Read_In,
            Check_In => WD_Check_In,
            Err_Flag => Err_Flag
```

```
    );
  A1: Reg port map (CLK, RegW, DR, SR1, SR2, Reg_In, ReadReg1,
ReadReg2);
  Imm_Ext <= x"FFFF" & Instr(15 downto 0) when Instr(15) = '1'
    else x"0000" & Instr(15 downto 0);  -- Sign extend immediate field
  DR <= Instr(15 downto 11) when Format = R
    else Instr(20 downto 16);           -- Destination Register MUX
(MUX1)
  ALU_InA <= ReadReg1;
  ALU_InB <= Imm_Ext when REGorIMM_Save = '1' else ReadReg2;   -- ALU
MUX (MUX2)
  Reg_in <= Mem_Bus when ALUorMEM_Save = '1' else ALU_Result_Save; --
Data MUX
  Format <= R when Opcode_State0 = 0 else J when Opcode_State0 = 2 else
I;
  Mem_Bus <= ReadReg2 when Writing = '1' else
    "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"; -- drive memory bus only during
writes
  ADDR <= PC when FetchDorI = '1' else ALU_Result_Save; --ADDR Mux
  IRAM_select <= FetchDorI;

  F_Code2 <= F_Code XOR ERR_MASK(35 DOWNTO 30);            -- Added for
Error Injection
  Opcode_State0 <= Opcode XOR ERR_MASK(29 DOWNTO 24);      -- Added for
Error Injection
  Opcode_State1 <= Opcode XOR ERR_MASK(23 DOWNTO 18);      -- Added for
Error Injection
  Opcode_State2 <= Opcode XOR ERR_MASK(17 DOWNTO 12);      -- Added for
Error Injection
  Opcode_State3 <= Opcode XOR ERR_MASK(11 DOWNTO 6);       -- Added for
Error Injection
  Opcode_State4 <= Opcode XOR ERR_MASK(5 DOWNTO 0);        -- Added for
Error Injection

  process(State, PC, Instr, Format, F_Code, F_Code2, opcode,
Opcode_State1, Opcode_State2, Opcode_State3, Opcode_State4, Op,
ALU_InA, ALU_InB,
          Imm_Ext, OpSave)
  begin
    FetchDorI <= '0'; CS <= '0'; WE <= '0'; RegW <= '0'; Writing <=
'0';
    ALU_Result <= "00000000000000000000000000000000";
    npc <= pc; Op <= jr; REGorIMM <= '0'; ALUorMEM <= '0';
      WD_Read_In <= "00";
    case state is
      when 0 =>  --fetch instruction
        nPC <= PC + 1; CS <= '1'; nState <= 1;
        FetchDorI <= '1';
      when 1 =>
        nState <= 2; REGorIMM <= '0'; ALUorMEM <= '0';
        if Format = J then
          nPC <= "000000" & Instr(25 downto 0); nState <= 0; --jump,
and finish
        elsif Format = R then  -- register instructions
          if    F_code2 = "100000" then Op <= add;   -- add

          elsif F_code2 = "100010" then Op <= sub;   -- subtract
```

```
          elsif F_code2 = "100100" then Op <= and1;  -- and
          elsif F_code2 = "100101" then Op <= or1;   -- or
          elsif F_code2 = "101010" then Op <= slt;   -- set on less
than
          elsif F_code2 = "000010" then Op <= shr;   -- shift right
          elsif F_code2 = "000000" then Op <= shl;   -- shift left
          elsif F_code2 = "001000" then Op <= jr;    -- jump register
          end if;
        elsif Format = I then -- immediate instructions
          REGorIMM <= '1';
          if Opcode_State1 = lw or Opcode_State1 = sw or Opcode_State1
= addi then Op <= add;
          elsif Opcode_State1 = beq or Opcode_State1 = bne then Op <=
sub; REGorIMM <= '0';
          elsif Opcode_State1 = andi then Op <= and1;
          elsif Opcode_State1 = ori then Op <= or1;
            -- ADDED INSTRUCTIONS --
              elsif Opcode_State1 = addiu then Op <= add2;
          end if;
          if Opcode_State1 = lw then ALUorMEM <= '1'; end if;
        end if;
      when 2 =>
WD_Read_In <= "01";    --WD_Check_In <= '1';
        nState <= 3;
        if   OpSave = and1 then ALU_Result <= ALU_InA and ALU_InB;
        elsif OpSave = or1 then ALU_Result <= ALU_InA or ALU_InB;
        elsif OpSave = add then ALU_Result <= ALU_InA + ALU_InB;
        elsif OpSave = sub then ALU_Result <= ALU_InA - ALU_InB;
        elsif OpSave = shr then ALU_Result <= ALU_InB srl
to_integer(numshift);
        elsif OpSave = shl then ALU_Result <= ALU_InB sll
to_integer(numshift);
        elsif OpSave = slt then -- set on less than
          if ALU_InA < ALU_InB then ALU_Result <= X"00000001";
          else ALU_Result <= X"00000000";
          end if;
            -- ADDED INSTRUCTIONS --
        elsif OpSave = add2 then ALU_Result <= UNSIGNED(ALU_InA) +
UNSIGNED(ALU_InB);

            -- END OF ADDED INSTRUCTIONS --
        end if;
       if ((ALU_InA = ALU_InB) and Opcode_State2 = beq) or
           ((ALU_InA /= ALU_InB) and Opcode_State2 = bne) then
          nPC <= PC + Imm_Ext; nState <= 0;
        elsif opcode_State2 = bne or opcode_State2 = beq then nState <=
0;
        elsif OpSave = jr then nPC <= ALU_InA; nState <= 0;
        end if;
      when 3 =>
        nState <= 0;
WD_Read_In <= "10";    --WD_Check_In <= '1';
        if Format = R or Opcode_State3 = addi or Opcode_State3 = andi
or Opcode_State3 = ori or Opcode_State3 = addiu then
          RegW <= '1';
        elsif Opcode_State3 = sw then CS <= '1'; WE <= '1'; Writing <=
'1';
```

```
        elsif Opcode_State3 = lw then CS <= '1'; nState <= 4;
        end if;
      when 4 =>
WD_Read_In <= "11";     --WD_Check_In <= '1';
       nState <= 0; CS <= '1';
       if Opcode_State4 = lw then RegW <= '1'; end if;
    end case;
  end process;

  process(CLK)
  begin
    if CLK = '1' and CLK'event then
      if rst = '1' then
        State <= 0;
        PC <= x"00000000";
            nState_OUT <= '0';
            Commit <= '0';
      else
            nState_OUT <= '0';
            Commit <= '0';
        State <= nState;
        PC <= nPC;
      end if;
      if State = 0 then Instr <= Mem_Bus;
        end if;
      if State = 1 then
         OpSave <= Op;
         REGorIMM_Save <= REGorIMM;
         ALUorMEM_Save <= ALUorMEM;
      end if;
      if State = 2 then ALU_Result_Save <= ALU_Result;
        end if;
            if nState = 0 then
                  nState_OUT <= '1';
                  Commit <= '1';
            end if;
    end if;
  end process;
end structure;
```

REFERENCES

[1] J. R. Srour, J. M. McGarrity, "Radiation effects on microelectronics in space", *Proceedings of the IEEE, vol. 76, pp. 1443-1469*, 1988.

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

[3] M. Santarini, "Cosmic radiation comes to ASIC and SOC design", *EDN*, 2005.

[4] Q. Zhou, K. Mohanram, "Transistor sizing for radiation hardening", *IEEE International Reliability Physics Symposium Proceedings, pp. 310-315, 25-29*, April 2004.

[5] R. C. Baumann, "The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction", *2002 International Electron Devices Meeting, pp. 329-332*, 2002.

[6] N. J. Warter, W.-m. W. Hwu, "A software based approach to achieving optimal performance for signature control flow checking", *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium, pp.442-449*, 26-28 June 1990.

[7] K. Seongwoo, A. K. Somani, "On-line integrity monitoring of microprocessor control logic", *Computer Design, 2001. International Conference on Computer Design Proceedings, pp.314-319*, 2001.

[8] R. G. Ragel, S. Parameswaran, "Hardware assisted pre-emptive control flow checking for embedded processors to improve reliability", *Hardware/software codesign and system synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th international conference , pp.100-105, 22-25*, Oct. 2006.

[9] J. Ohlsson, M. Rimen, U. Gunneflo, "A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog", *Twenty-Second International Symposium on Fault-Tolerant Computing, Digest of Papers,pp.316-325*, 8-10 July 1992.

[10] W. H. Robinson, M. L. Alles, T. A. Bapty, B. L. Bhuva, J. D. Black, A. B. Bonds, L. W. Massengill, S. K. Neema, R. D. Schrimpf, J. M. Scott, "Soft Error Considerations for Multicore Microprocessor Design", *IEEE International Conference on Integrated Circuit Design and Technology*, June 2007.

[11] R. Baumann, "Soft errors in advanced computer systems", *IEEE Design & Test of Computers, vol.22, no.3, pp. 258-266*, May-June 2005.

[12] S. S. Mukherjee, J. Emer, S. K. Reinhardt, "The soft error problem: an architectural perspective", *Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture*, 2005.

[13] M. Maniatakos, N. Karimi, Y. Makris, A. Jas, C. Tirumurti, "Design and Evaluation of a Timestamp-Based Concurrent ErrorDetection Method (CED) in a Modern Microprocessor Controller", *IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems, pp.454-462*, 1-3 October 2008.

[14] R. W. Hamming, "Error detecting and error correcting codes", *BellSystem Tech. J, vol. XXVI, no. 2, pp. 147-160*, April 1950.

[15] R. E. Lyons, W. Vanderkulk, "The Use of Triple-Modular Redundancy to Improve Computer Reliability", *IBM Journal*, April 1962.

[16] V. D. Agrawal, C. R. Kime, K. K. Saluja, "A Tutorial on Built-in Self-Test. I. Principles", *IEEE Design Test 10, 1, 73-82*, Jan. 1993.

[17] A. Sanyal, S. M. Alam, S. Kundu, "A Built-In Self-Test Scheme for Soft Error Rate Characterization", *14th IEEE International On-Line Testing Symposium, 2008, pp.65-70*, 7-9 July 2008.

[18] J. B. Eifert, J. P. Shen, "Processor Monitoring Using Asynchronous Signatured Instruction Streams", *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 'Highlights from Twenty-Five Years', pp.106-, 27-30*, June 1995.

[19] C. Roth, Jr., L. K. John, "Digital Systems Design Using VHDL", 2008.

[20] R. P. Weicker, "Dhrystone benchmark: rationale for version 2 and measurement rules", *SIGPLAN Not. 23, 8, 49-62*, August 1988.

[21] N. Pinckney, T. Barr, M. Dayringer, M. McKnett, N. Jiang, C. Nygaard, D. M. Harris, J. Stanley, B. Phillips, "A MIPS R2000 IMPLEMENTATION", *Proceedings of the 45th Annual Conference on Design Automation, 102-107*, June 08 - 13, 2008.

[22] M. Hsueh, T. K. Tsai, R. K. Iyer, "Fault Injection Techniques and Tools", *Computer 30, 4, 75-82*, April 1997.

[23] F. Lima, S. Rezgui, L. Carro, R. Velazco, R. Reis, "On the Use of VHDL

Simulation and Emulation to Derive Error Rates", *Radiation Effects on Components and Systems Conference (RADECS) Proceedings*, 2001.

[24] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P.D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, R. Jenkal, "FreePDK: An Open-Source Variation-Aware Design Kit", *IEEE International Conference on Microelectronic Systems Education, pp.173-174*, 3-4 June 2007.