May 2001

UILU-ENG-01-2210
CHRC-01-01

A PORTABLE SOFTWARE TOOL FOR MEASUREMENT OF TRANSIENT ERRORS IN COMMERCIAL MICROPROCESSORS

Last Copy

Karen E. Wells and Janak H. Patel

Do Not Take

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street, Urbana, IL 61801

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE **May 2001**	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE **A Portable Software Tool for Measurement of Transient Errors in Commercial Microprocessors**	5. FUNDING NUMBERS **NASA-JPL-1215699**
6. AUTHOR(S) **Karen. E. Wells and Janak. H. Patel**	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Coordinated Science Laboratory** **University of Illinois at Urbana-Champaign** **1308 West Main Street** **Urbana, Illinois 61801-2307**	8. PERFORMING RGANIZATION REPORT NUMBER **UILU-ENG-01-2210** **(CRHC-01-01)**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) **Jet Propulsion Laboratory** **4800 Oak Grove Drive** **Pasadena, CA 91109**	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official position, policy or decision, unless so designated by other documentation

12a. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited.**	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research describes a software tool that was written to support NASA's Remote Exploration and Experimentation (REE) project. The REE project attempts to improve the reliability of commercial off-the-shelf (COTS) microprocessors by supporting research devoted to bringing commercial supercomputing technology into space. COTS microprocessors are typically not radiation hardened and are therefore susceptible to transient errors due to alpha-particle radiation found in space environments.

The software tool developed for this research runs on a microprocessor to detect and measure the error rate of a processor exposed to radiation. It will estimate the location of any errors encountered and output information on these errors. The program is written in high-level language C and is broken down into separate routines for each testable hardware unit in a microprocessor. The tool is designed to be portable across several different platforms and is intended to provide crucial information about the state of system. Using the physical fault injection technique of power supply disturbances, results are presented that verify the tool is capable of detecting errors within the processor.

14. SUBJECT TERMS Error detection, SEU, Microprocessors, Control flow error, Fault injection, Error rate, Error latency	15. NUMBER OF PAGES 62 16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT **UNCLASSIFIED**	18. SECURITY CLASSIFICATION OF THIS PAGE **UNCLASSIFIED**	19. SECURITY CLASSIFICATION OF ABSTRACT **UNCLASSIFIED**	20. LIMITATION OF ABSTRACT **UL**

A Portable Software Tool for Measurement of Transient Errors in Commercial Microprocessors

Karen E. Wells and Janak H. Patel
University of Illinois at Urbana-Champaign

ABSTRACT

This research describes a software tool that was written to support NASA's Remote Exploration and Experimentation (REE) project. The REE project attempts to improve the reliability of commercial off-the-shelf (COTS) microprocessors by supporting research devoted to bringing commercial supercomputing technology into space. COTS microprocessors are typically not radiation hardened and are therefore susceptible to transient errors due to alpha-particle radiation found in space environments.

The software tool developed for this research runs on a microprocessor to detect and measure the error rate of a processor exposed to radiation. It will estimate the location of any errors encountered and output information on these errors. The program is written in the high-level language C and is broken down into separate routines for each testable hardware unit in a microprocessor. The tool is designed to be portable across several different platforms and is intended to provide crucial information about the state of the system. Using the physical fault injection technique of power supply disturbances, results are presented that verify the tool is capable of detecting errors within the processor.

Key Words: Error detection, SEU, Microprocessors, Control flow error, Fault injection, Error rate, Error latency

This material is based upon work supported by the NASA-JPL under Award No. 1215699

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

This research describes a software tool that was written to support NASA's Remote Exploration and Experimentation (REE) project. The work was supported by the NASA Jet Propulsion Laboratory (JPL).

1.1 Background

The prevention of system failure in space applications is extremely important. The consequences of failures can be costly and hazardous. Of particular concern are temporary errors, specifically, those due to transient faults. Transient faults are induced by external perturbation such as power supply fluctuations or radiation. In space applications, they are predominantly caused by alpha particle radiation. As a result, many companies aim to produce radiation-hardened systems to be used in space programs. However, these systems are often more costly, higher in volume, higher in mass and power consumption, and lower in performance than components presently available off-the-shelf. In particular, today's microprocessors that are available commercially have many advantages over those in radiation-hardened systems. Commonly referred to as commercial off-the-shelf (COTS) components, these microprocessors are readily available, of high packaging density, cheap, capable of supporting extensive software, and consume relatively low power [1]. With the increased use of software in space and the drive toward smaller, faster, better, and more cost-effective satellites, COTS systems are attractive for use in computer-based satellite systems. However, since COTS microprocessors are not radiation-hardened, they are much more vulnerable to transient failures [2]. As a result, various tools and methodologies are being researched to tolerate the effects of radiation on COTS microprocessors in space applications.

The NASA Remote Exploration and Experimentation (REE) project attempts to improve the reliability of COTS microprocessors by supporting research devoted to bringing commercial supercomputing technology into space. The design of highly reliable computer systems requires a thorough understanding of errors in the underlying COTS components, of which a microprocessor is the principal part. There are many factors that make a microprocessor more susceptible to radiation induced upsets. However, there is currently little data for analyzing and detecting transient errors in microprocessors. The research described in this paper is part of the REE project and employs a new method for detecting errors in a microprocessor due to radiation. A high-level software tool has been created to detect and characterize radiation-induced errors in space borne applications running on COTS components.

1.2 Software Tool Description

The software tool developed for this research runs on a microprocessor to measure the error rate and estimate the location of any errors encountered. It is broken down into separate routines for each testable hardware unit in a microprocessor and it keeps count of the number of errors found in each unit. In this research, a testable hardware unit can be defined as a collection of circuitry where most transient errors occurring in that circuitry can be detected and attributed to that circuitry in a high-level language. Typically, a microprocessor is made up of many hardware units, several of which are testable. Chapter 5 describes in further detail the general hardware units of a microprocessor and which ones are considered to be testable from a high-level language.

Each routine in the program uses a sequence of computations that are performed millions of times in a loop as a way of testing the specific hardware. The operations in each routine are chosen in such a way as to maximize the switching activity in the corresponding circuitry. Also, the error latency in a given unit is minimized by flooding it with millions of operations per second. This increases the error coverage by utilizing as much of the circuitry as possible as fast as possible.

There are many advantages of this self-monitoring software tool for error measurements. It has excellent potential for accurate localization of a single-event upset (SEU) to within a small functional block. Errors found by this tool will have a latency of only a few instruction cycles in most cases. Also, the probability of capturing an SEU is very high with the use of enhanced signal activities and the creation of sensitized paths. The tool is portable to different systems and can be used during long space missions to periodically measure the error rates. The error rates can vary during a long mission due to a changing environment and possibly due to aging of the electronics on the chip. Such a tool can give an early warning to the system of increased error rates.

Some assumptions are made for the tool to work successfully. The type and frequency of the errors are assumed to be such that the error rate is not so high as to crash the monitoring tool very frequently. The only data in that case would be the system crash rate, not an error rate. The errors that do not cause an immediate crash are the most worrisome. They are also far more likely than catastrophic errors, and if they are not detected early, they can multiply and lead to a system crash or to additional new errors. For these reasons, the most interesting case for measurement is for infrequent SEUs.

2 RELATED WORK

Determining the suitability of COTS microprocessors for space applications is a subject of ongoing research. Normally, this involves predicting the expected single-event upset (SEU) rate in the microprocessor when it is flown in space. This is generally done using ground-based radiation tests where the device to be tested is exposed to an appropriate particle beam, and subsequently checked to determine if there have been any state changes. The results from these tests are used to predict the underlying SEU rate of the device. Over the last several years, many ideas related to testing the effects of SEUs on microprocessors have been presented:

- In [2], software tools are presented for predicting the rate and nature of observable SEU-induced errors in microprocessor systems. These tools are built around a commercial microprocessor simulator and are used to analyze real satellite application systems.
- In [3], the impact of physical fault injection of transient faults on processor execution is assessed. The fault injection is based on two complementary methods using (1) heavy-ion radiation, and (2) power supply disturbances. Approximately 12,000 transient faults were injected into the target microprocessor, a Motorola MC6809E 8-bit CPU, running three different workloads. Three error-detection mechanisms (two software-based mechanisms and one watchdog timer) were combined and used to detect as many errors as possible.
- Also, in [4], two generations of the 80Cx86 microprocessor family and two floating-point digital signal processors (DSP) were tested for single-event effects (SEE). The SEE rates for each of the device types were measured and the upset vulnerability was correlated to the size of the program running on each processor.
- In [5], the SEU rates of the Pentium MMX and Pentium II microprocessors using proton irradiation are determined. An evaluation of the performance of these microprocessors in the space radiation environment is also presented. Test software was used to determine the SEU behavior of the ALU, FPU, registers, and cache memory functional blocks of the microprocessors. A comparison is given of each processor's susceptibility to proton irradiation in various modes of operation.

- Similarly, in [6], SEE test results are given for the Intel 80386 family and the 80486 microprocessor. Both single-event upset and latchup conditions were monitored.
- Another study by Asenek [7] predicts the rate of observable SEU-induced errors in a microprocessor system by measuring the errors in terms of the SEU application cross-section as opposed to the SEU cross-section as a function of the device's technology only. A software tool is used to do the predictions as the microprocessor executes real application software. Results obtained from simulating the nature of SEU-induced errors are shown to correlate with ground-based radiation test data.

Although this work offers useful information on the susceptibility of a particular microprocessor to transient errors, the data is still somewhat limited. They all present data for specific microprocessors and do not attempt to localize the errors. Also, none of the tools is portable to other systems. It is more desirable to produce results that can be applied to any system and to run the tools on any microprocessor, which is what this research attempts to do.

Another area of related work is that done by the NASA Remote Exploration and Experimentation (REE) project itself. The REE Project is working to incorporate a custom, but architecturally insensitive, software implemented fault tolerance (SIFT) middleware layer, as well as a generic library of algorithm-based fault tolerance (ABFT) techniques, to enable the direct use of latest generation commercial hardware and software components in future space systems. This strategy will allow high throughput computation even in the presence of relatively high rates of radiation-induced transient upsets as well as in the presence of permanent faults. Previous work in this area includes:

- A first-generation testbed presented in [8] is created to test the above concepts. The testbed is equipped with fault injection capabilities and is constructed out of COTS hardware and software. The methodology used to develop a detailed radiation fault model for the REE testbed architecture is discussed. Also, result checking is suggested as a way of enforcing hardware/software reliability. Result checking relies on developing tests that can confirm the validity of operation output. Their work focuses on computation errors inherent in the system, rather than on environmentally induced faults.
- Huang and Abraham [9] introduce ABFT—a technique that encodes matrices using checksum matrices. These are then used to detect and correct faults in matrix

operations. They address matrix operations performed using processor arrays with regard to detecting errors generated by a faulty processor within the array.

- Jou and Abraham [10] present an ABFT error detection scheme for FFT networks. The method employs an encoding and decoding scheme to detect single errors.

Other work at the microprocessor level have primarily focused on vulnerability assessment and software detection methods:

- In [11] a detailed analysis of the vulnerability of the Z80 microprocessor, based on ion-bombardment testing, is presented.
- In [12], a simulation tool called DYNAMO is used to study the effects of transient faults in large digital circuits.
- In [13], a series of experiments aimed at error analysis through the physical insertion of faults have been conducted at the NASA AIRLAB test-bed facility. An experimental analysis is presented to study the susceptibility of a microprocessor-based jet engine controller to upsets caused by current and voltage transients.
- In [14], a hierarchical error detection framework for a software implemented fault tolerance (SIFT) layer is proposed for a distributed system. A four-level error detection hierarchy is proposed in the context of Chameleon, a software environment for providing adaptive fault-tolerance in an environment of COTS system components and software. The design and implementation of a software-based distributed signature-monitoring scheme is described.
- In [15], the design and evaluation of a preemptive control signature technique (PECOS) for on-line detection of control flow errors is presented. The technique uses assertions that can be embedded in the assembly language code and that are triggered by control flow instructions in the code. PECOS can detect errors in control flow that spans multiple subroutines or source files as well as control-flow, which is determined at runtime. Software-based error injection techniques are used to evaluate PECOS.
- In [16], PECOS is used in combination with a data audit subsystem to eliminate fail-silence violations, reduce the incidence of crashes, and eliminate hangs. Software-implemented error injection is performed to evaluate the system.

All of the above studies are based on low-level hardware-specific techniques. The research described in this paper differs in that the tool used to detect and locate transient errors is portable across multiple platforms and can be used in a multitude of setups. The goal in using COTS microprocessors is to reduce cost and increase performance. Our tool supports that by providing error analysis while not limiting the type of microprocessors or systems to be used.

3 TRANSIENT ERROR OVERVIEW

A transient error is an error that can appear and disappear within a very short period of time. They are temporary or nonpermanent and may arise in a working circuit during its operation due to a variety of noise sources [12]. For example, internal noise sources such as power transients and capacitive and inductive crosstalk are common causes, as well as external noise sources such as cosmic particle hits (e.g., alpha particles) [6]. Cosmic particles are the common cause of transient errors in space applications; therefore, it is the alpha particle hits that are of most interest in this work. However, the other sources of transient errors can be used for test purposes.

3.1 Single-Event Upsets

A single-event upset (SEU) is a type of transient error that is commonly caused by alpha particle radiation. A SEU is a soft error introduced when an ionizing heavy ion penetrates the depletion region of a reversed-biased pn-junction. The heavy ion affects only stored information by changing bit values from 0 to 1 or vice versa. The majority of heavy ions affect only single bits. No hardware is damaged by the ions. The SEUs are assumed to occur randomly both in time and in position within a circuit. Typically the transient introduced by an alpha particle lasts only for a very short period of time, on the order of a fraction of a nanosecond [3].

Transient faults and SEUs are produced in today's microprocessors in several ways. Such things as lower supply voltages and lower transistor threshold values make a microprocessor more susceptible to transient errors. Also, lower noise margins, current leakage, and charge injections in dynamic circuits contribute to the susceptibility of microprocessors to SEUs. The higher density of interconnects and the high slew rate of logic signals together make crosstalk due to capacitive coupling a serious problem. Some circuit defects such as resistive shorts and opens are missed by the manufacturing tests.

3.2 Analysis of Transient Faults

It has been reported that transient faults account for 80% or more of the failures in digital systems [12]. The effect of these transient faults is to change the behavior of the digital circuit in some unexpected manner, often producing incorrect results. If the digital system is a critical application, such as a biomedical, space, military, or avionic application, the results could be catastrophic. If the critical areas in the circuit that are more susceptible to transient-fault disturbances are identified, appropriate actions may be taken to prevent those catastrophes.

To analyze the nature of SEU-induced errors in microprocessor systems, it is useful to predict the rate of observable SEU-induced errors in microprocessor systems and to identify and classify the errors into different categories. It is also useful to understand whether the occurrence of an SEU will cause a system to fail catastrophically or to simply lose service for short duration. This is essential in justifying its suitability for a particular application or mission.

SEUs affecting data are associated with upsets in the execution units such as the ALU, FPU, or data cache. SEUs affecting control result in program flow errors (sequencing errors), upsets in the instruction fetch unit or the branch prediction unit. These are more severe than data errors since SEUs affecting control would be expected to cause an exception. However, SEUs affecting data are particularly troublesome because they typically have fewer obvious consequences than an SEU affecting control. It should be noted that since memory will be error detecting and correcting, faults to memory would be largely screened; most data faults will therefore affect the microprocessor or its cache.

In [7], results were obtained from simulating the nature of SEU-induced errors in a microprocessor as well as results from radiation tests causing SEUs. The experiments in this paper used an 8052 microprocessor system executing the COTS2 operating system program [7]. For the radiation tests, of the 131 observed errors, approximately 31% were due to transient errors, 64% were resets, 5% were propagated errors. This data is useful because it gives us a feel for the type and frequency of errors we should expect in a microprocessor exposed to radiation.

3.3 Fault Models

Fault models are typically used to help us characterize the behavior of errors. We make the assumption that faults behave according to some fault model. This allows us to specifically define the types of faults that will be considered and the behavior these faults will have [14]. Also, we can represent the behavior of physical occurrences. Several defects are usually mapped to one fault model. Fault models are not 100% accurate, but they make problems tractable and represent the types of faults that can occur. The general fault model used for transient faults in this research includes incorrect signal values caused by coupled disturbance. These disturbances include those due to particle irradiation. The fault models and types of errors expected for specific microprocessor units are presented in Chapter 5.

Since the software tool designed in this project is intended for use in space applications where radiation exposure is imminent, a radiation fault model proves to be helpful. In a non-radiation hardened system, faults due to radiation are not prevented, but rather allowed to occur. The error (the logical manifestation of the fault as seen by the system or software) or a subsequent manifestation due to propagation of the error is then detected and handled by the software [8]. Clearly, the design of such a system is dependent on a detailed understanding of the types of faults that will occur, the errors generated by these faults, and the rate at which they will appear.

It is useful to predict what types of errors will be generated, under what conditions the system will become bogged down in error handling, and under what conditions errors will propagate through the system error detection and containment boundaries. It is for this reason we have developed the software tool described in this paper. Its ultimate purpose is to output a real-time error rate so that decisions can be made on the vulnerability of the system as well as individual components of the processor.

The principle faults for the REE environments are single bit flips; however, there is an expectation that multiple bit flips will become more prevalent due to shrinking feature sizes [8]. In this fault mode, several physically adjacent cells may be disrupted by the passage of a single energetic particle. However, these single-event multiple upsets are still relatively rare and for this research, they are not considered.

From a high-level point of view, faults due to radiation can be grouped into two types:

- *Latch faults* include latches, flip-flops, memory cells, and any other structure that persistently stores a bit. 'Visible' latches are included, such as data registers, as are 'invisible' latches, which are used to implement structures such as instruction pipeline stages and processor register reservation scoreboards [8].
- *Gate faults* occur when a SEU happens at approximately the same time as a clock transition, thus causing the gate to flip its effective bit value [8].

Due to the tight timing required for a gate fault to propagate to, and be latched by, a register, faults in latches are tens of thousands of times more likely than faults at the gate level, or combinatorial logic. As the clock rate increases, this difference will shrink due to the increased fraction of time available for combinatorial logic to present erroneous values to registers, which may then latch these transients [8].

The goal of this section is to provide a basic idea of what types of faults are expected from radiation and how these faults relate to errors that may be produced. More information on these types of faults and the fault model structure can be found in [8]. The types of errors expected in specific functional blocks of a microprocessor from this fault model can be found in Chapter 5. Also, the functional modules of each unit are examined to determine the effects of errors on their behavior.

4 SOFTWARE ERROR DETECTION TECHNIQUES

Many software fault detection techniques have been implemented in the past. However, all of these techniques attempt to change the original application code in some way, or they are coded completely in machine language, which makes them very difficult to port to other systems. Also, they are often nondeterministic in that it is unknown what instruction sequences will be executed and what the results of various operations in the application will be at any given time. The tool described in this thesis is a stand-alone program that was written from scratch in a high-level language, therefore making it deterministic as well as portable. It is known in advance what instructions will be executed and what the results of those instructions should be. This allows the use of some common software error detection techniques such as time redundancy and consistency checks to detect the transient errors occurring in a COTS microprocessor.

4.1 Time Redundancy

Time redundancy is a method that uses the repetition of computations in ways that allow errors to be detected. Figure 1 illustrates this idea with a block diagram. The fundamental concept is to perform the same computation two or more times and compare the results to determine if a discrepancy exists. If an error is detected, the computations can be performed again to see if the disagreement remains or disappears [17]. If the discrepancy remains, then there is a permanent error present; however, if the discrepancy disappears, then a transient error has occurred.

4.2 Consistency Checks

Consistency checks are used as another means of capturing transient errors in software. They use a priori knowledge about the characteristics of information to verify the correctness of that information. For example, when adding two known numbers, such as 2 and 3, we know that the result should be 5. If the result is not 5 at the moment it is checked, then some sort of error is present. Combining this idea with the time redundancy method above, the addition can be performed several times, and if the result is consistently something other than 5, there is most likely a permanent error present. However, if the result is 5 most of the time and something other than 5 the rest of the time, then there is likely a transient error occurring.

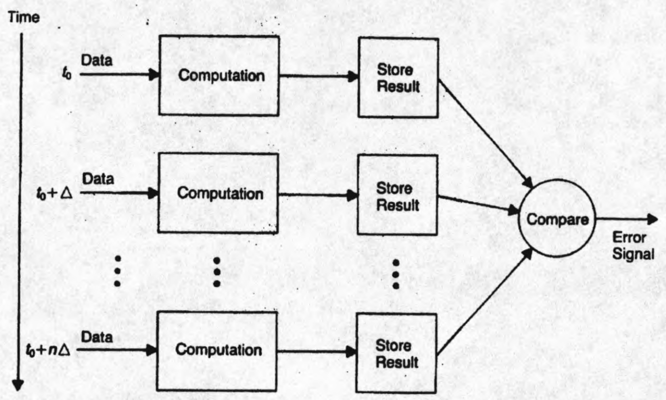

Figure 1: Time Redundancy Example.

The software program described in this paper uses consistency checks as the primary methods of detecting errors. A loop consisting of a group of operations that are checked for correctness upon completion is executed millions of times. Figure 2 illustrates the basic flowchart used for each routine when testing the functional units of the microprocessor.

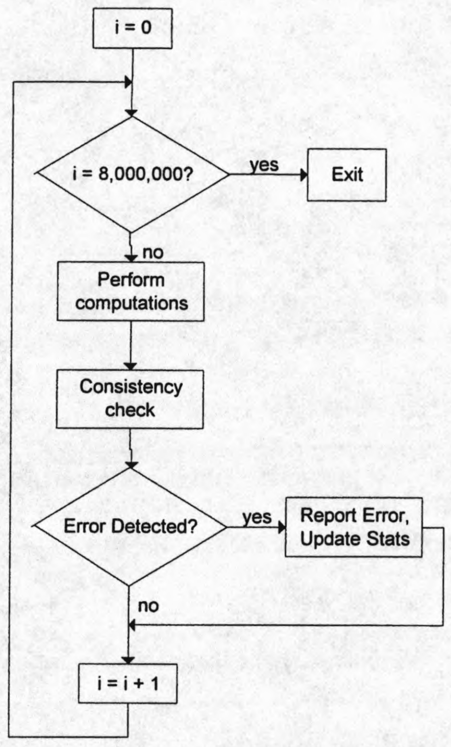

Figure 2: Flowchart for Functional Unit Routines.

5 PROCESSOR MODEL

To aid in localizing any errors encountered, the microprocessor is represented in terms of its functional hardware units. A fault model is then developed for each of these functional units. In this way, computations can be generated to test each unit without the detailed knowledge of how the instruction sequencing and control section for each unit are implemented. Also, some units can be broken down into smaller units if it is possible to localize errors to smaller areas in the circuitry. For example, the integer unit can be broken down into an add unit, a subtract unit, a multiply unit, etc., because they all use different hardware that can be tested explicitly from a high-level program. However, some units, such as the branch processing unit, cannot be broken down further because it is impossible to write a high-level computation that will exclusively test its subunits such as the branch prediction hardware.

Also, for portability purposes, some basic assumptions are made about the processor when developing the software tool. First, it is assumed that the registers are of size 32 bits. Second, it is assumed that certain execution and control units exist in all COTS processors. It is understood that there are other units unique to specific processors, for example a vector unit; however, these are not targeted directly in this version of the software tool. Third, because there are many ways to implement each functional unit, we will concentrate on obtaining high switching on the inputs and outputs and assume that this will maximize the activity in any given design.

The execution units assumed to be common to all COTS processors as well as their subunit circuitry that can be tested are as follows:

- Integer units 1 and 2
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Logical AND
  - Logical OR
  - Logical XOR
- Register unit

- Floating point unit
    - Addition
    - Subtraction
    - Multiplication
    - Division
- Data cache unit
- Load/store unit

The control units assumed to be common to all COTS processors are:

- Instruction fetch unit
- Branch processing unit

The instruction fetch unit encompasses components such as the instruction cache, the instruction register, the dispatch unit, and any other hardware involved in fetching instructions. The branch processing unit includes hardware such as the branch target instruction cache, the branch history table, control registers, and any other circuitry involved in branch prediction.

There are other units not mentioned above that are extremely difficult to test from a high-level program and are therefore left out when generating the test routines. For example, detecting errors occurring in the memory management unit (MMU), the reservation stations, the completion unit, and the bus interface unit cannot be thoroughly and exclusively accomplished in a high-level program. Also, differentiating between errors in the L1 and L2 cache is difficult. But, even though errors in these units are not tested for explicitly, many other operations that are performed will utilize these units, and if errors occur, it is possible that they will be detected indirectly.

The preliminary version of the tool developed in this research was written for a Motorola PowerPC processor – the MPC750. A block diagram of this processor is shown in Figure 3. The MPC750 processor contains all of the units described previously and provides a basis for the software tool. The results presented in Chapter 12 are gathered from testing the tool on the MPC750.

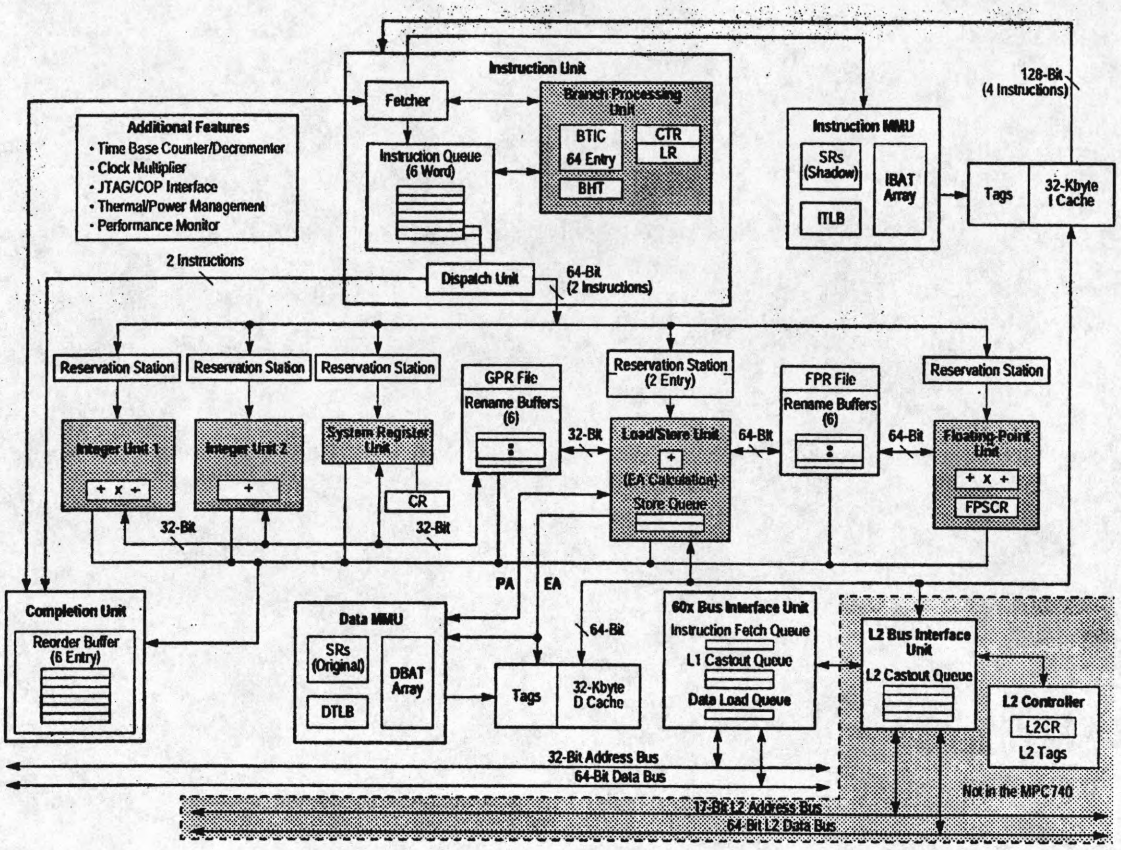

Figure 3: MPC750 Processor Block Diagram.

The following chapters present more detail on how the testing was done for each unit. However, it should be pointed out that the work for this research was split between two students and, as a result, only certain units are discussed in this thesis. For a detailed description on how the data cache and the floating point unit were tested, please refer to the thesis to be written by Hari Kommaraju [18].

6 TESTING THE INTEGER UNITS

It is assumed that any COTS processor used in the space applications will have at least two integer units. Typically, these integer units have slightly different purposes. In the MPC750 processor shown in Figure 3, integer unit 1 is used for all integer computations, whereas integer unit 2 is used only for addition. There may be other integer units in other processors, but we will assume that it is sufficient to test for two.

In general, integer unit 1 is assumed to encompass the following operations:

- Addition
- Subtraction
- Multiplication
- Division
- Logical AND
- Logical OR
- Logical XOR

Each of the above operations uses different logic within the integer unit and each requires a different operational code (opcode) value in the corresponding instruction. Any errors found while performing the above operations can be attributed to the particular logic for that operation.

Integer unit 2 is assumed to provide only addition. The specific details of how the processor decides which integer unit an addition instruction will be allocated to are unknown. As a result, integer unit 2 must be tested in conjunction with integer unit 1. To guarantee that integer unit 2 is utilized, integer unit 1 must be busy. This is accomplished by issuing integer multiplies in between integer additions. Since integer unit 2 can only do additions, the multiplications are sent to integer unit 1 and the subsequent additions will be sent to integer unit 2.

Any error in the integer units can be covered with the appropriate fault model. Each integer operation can be broken down into several identical logic gates that can be referred to as cells. Each cell represents the computation for one "slice" of the data. For example, the adder cell performs the addition of 3 bits and the logical AND cell performs the AND of 2 bits. The general fault assumption at the cell level is that a faulty cell can change its behavior in any arbitrary way, as long as it remains a combinational circuit [19]. Also, an appropriate cell fault model can cover any fault in the intercell array.

The operations used for each integer routine are either independent or dependent computations. The independent computations are executed in a separate loop from that of the dependent computations. Independent computations are executed one after another, and then the consistency checks are performed after the last computation completes. The idea is to fill up the arithmetic pipeline by flooding it with operations. This will exercise the pipeline hardware and control, and it will indirectly check for some types of control flow errors. If the program erroneously jumps from the middle of the computations to the middle of the consistency checks, then the next consistency check will fail.

The dependent computations are performed so that one operation must complete before the next one can execute. Several operations are performed in a row, and then a single consistency check is performed at the end. These types of instructions will introduce bubbles in the pipeline and exercise different pipeline control than that of the independent computations. They will also check for some types of control flow errors. If the program erroneously jumps from the start of the dependent computations to the end of the dependent computations, then the final consistency check will fail.

The pipeline control for independent and dependent instructions is different and can be exercised by performing the independent and dependent operations described above. Dependent operations in a pipeline typically require more complex control than independent operations, and this is a good way to check the functionality of that hardware. Code examples for the independent as well as the dependent operations can be found in most of the sections that follow. The logical operations are only tested with independent computations because they typically do not take many clock cycles to execute.

6.1 Addition

The fault model used for integer addition is the multiple cell fault model (MCFM), which means that all of the cells can be faulty [19]. In the MCFM, an adder can be tested by a minimum test set of size 11 independent of the number of cells in the array. An example of the minimum test set is given in Table 1. Each test vector has a periodic pattern. For any arbitrary adder, the test set in Table 1 tests every cell with all input combinations. If there is a faulty cell, at least one of the two sum outputs of this faulty cell and its next cell will be different from the correct value. Therefore, this test set is sufficient to completely test any ripple carry adder under MCFM. Consistency checks in the program are used to catch these errors. Figure 4 shows a code excerpt with the independent operations along with the consistency checks. Figure 5 also shows a small code excerpt with some of the dependent additions where each subsequent addition depends on the previous result and a final check is performed at the end.

Table 1: Minimum Test Set for Ripple Carry Adders Under MCFM [19].

vector	c_0	x_0y_0	x_1y_1	x_2y_2	x_3y_3	x_4y_4	x_5y_5	. . .
t1	0	00	00	00	00	00	00	. . .
t2	0	00	01	00	01	00	01	. . .
t3	0	01	00	01	00	01	00	. . .
t4	0	00	10	00	10	00	10	. . .
t5	0	10	00	10	00	10	00	. . .
t6	0	00	11	00	11	00	11	. . .
t7	0	11	00	11	00	11	00	. . .
t8 (t8')	1 (0)	00 (10)	00	11	00	00	11	. . .
t9 (t9')	1 (0)	01 (11)	01	01	01	01	01	. . .
t10 (t10')	1 (0)	10 (11)	10	10	10	10	10	. . .
t11 (t11')	1 (0)	11	11	11	11	11	11	. . .

Note that in Table 1, control over the carry bit, c_0, is required in some of the vectors (t8 – t11). However, from a high-level program, it is difficult, if not impossible, to explicitly set the carry-in bit. As a result, these vectors must be modified. The x_0y_0 bits for vectors t8-t11 are modified so that the carry will be rippled through as intended. The new vectors are labeled t8'-t11' and represent the changes made in parenthesis shown in the table.

```
CheckIntegerAddition()
 register int R0, R1;

 for (int i=0; i < LOOP_ITERATIONS; i++) {
   R0 = 0x00000000 + 0x00000000;  // vector t1
   R1 = 0x00000000 + 0xaaaaaaaa;  // vector t2
      .
      .
      .
   if (R0 != 0xaaaaaaaa) UpdateStats(Err, ia);
   if (R1 != 0xffffffff) UpdateStats(Err, ia);
      .
      .
      .
 }
```

Figure 4: Example Code for Independent Integer Addition Operations.

```
CheckIntegerAddition()
 register int R0;

 for (int i=0; i < LOOP_ITERATIONS; i++) {
   R0 = 0x00000000;
   R0 = R0 + 0x0fffffff;
   R0 = R0 + 0x0fffffff;
   R0 = R0 + 0x0fffffff;
      .
      .
      .
   if (R0 != 0xfffffff0) UpdateStats(Err, ia);
 }
```

Figure 5: Example Code for Dependent Integer Addition Operations.

6.2 Subtraction

The fault model used for subtraction is the same as that for addition. Also, the same functional vectors are used. A separate routine is used for subtraction because subtraction involves one more step than addition – a 2's complement step – and the instruction code for subtraction will be different. The common implementation for subtraction is to take the 2's complement of the subtrahend and add it to the minuend.

It is possible that the subtraction hardware uses hardware for addition to generate the result. It is therefore recommended that the routine for addition be executed immediately before the routine for subtraction. If no errors are present in the addition routine, then any errors occurring in the subtraction routine are most likely due to the extra hardware required for subtraction.

6.3 Multiplication

There are several algorithms and hardware designs for implementing multiplication. The goal in this program is to use vectors that will maximize the switching of the multiplication steps. Ideally, we would like every bit in the multiplicand, the multiplier, and the product to change value during each cycle. However, this is difficult to do without knowing the exact hardware implementation in the system. The best that we can do is to choose operands that maximize switching at the inputs and also at the outputs. Table 2 shows the vectors used for integer multiplication along with the number of bits that are switching in the inputs and outputs from one multiplication to the next. To clarify, between the first and second multiplies, i.e., lines 1 and 2 in the table, 16 bits switch in the first input, 16 bits switch in the second input, and 16 bits switch in the output. Both signed and unsigned numbers of the same magnitude are used.

Table 2: Vectors for Integer Multiplication.

Multiplicand	Input Bits Switching	Multiplier	Input Bits Switching	Product	Output Bits Switching
0x00000000		0x00000000		0x00000000	
0x0000ffff	16	0x0000ffff	16	0xfffe0001	16
0x00000000	16	0xaaaaaaaa	16	0x00000000	16
0xffffffff	32	0x00000001	17	0xffffffff	32
0x00005555	24	0x00005555	7	0x1C718e39	17
0x00000001	7	0xffffffff	24	0xffffffff	17
0x0000aaaa	9	0x0000aaaa	24	0x71c638e4	17
0x00004924	9	0x00004924	9	0x14e58d10	17
0x0000ffff	11	0x00000002	15	0x0001fffe	16
0x00009249	10	0x00009249	7	0x539758D1	18
0x00002492	11	0x00002492	11	0x05396344	18

Also, dependent multiplications are performed where each subsequent multiply depends on the previous result and a final check is performed at the end. Table 3 shows the computations for each step, each intermediate value, the final result expected, and the number of bits switching in the output between each computation. While these methods do not accomplish perfect switching among the bits for each output, they are satisfactory in attempting to catch errors in the multiplication loop.

Table 3: Dependent Integer Multiplication Vectors.

Multiply Operation	Intermediate result	Bits Switching in the Output
`UiR0 = 0x0000000f;`	0x0000000f	
`uiR0 = uiR0 * 0x0000001b;`	0x00000195	5
`uiR0 = uiR0 * 0x0000001b;`	0x00002ab7	5
`uiR0 = uiR0 * 0x0000001b;`	0x0004814d	12
`uiR0 = uiR0 * 0x0000001b;`	0x0079a31f	11
`uiR0 = uiR0 * 0x0000001b;`	0x0cd43445	16
`uiR0 = uiR0 * 0x00000002;`	0x19a8688a	18
`uiR0 = uiR0 * 0x00000002;`	0x3350d114	18
`uiR0 = uiR0 * 0x00000002;`	0x66a1a228	18
`uiR0 = uiR0 * 0x00000002;`	0xcd434450	18
Final Result	0xcd434450	

6.4 Division

Division involves several of the same issues addressed in creating the vectors for multiplication. Again, it is difficult to maximize the switching in the hardware without knowing the exact implementation. The vectors are chosen so that the bits in the dividend, divisor, and quotient are switching as frequently as possible in a high-level program. Table 4 shows these vectors along with the number of bits that are switching in the inputs and outputs from one divide to the next. Both signed and unsigned integers with the same magnitude are tested.

Table 4: Vectors for Integer Division.

Dividend	Input Bits Switching	Divisor	Input Bits Switching	Quotient	Output Bits Switching
0x00000000		0xffffffff		0x00000000	
0xffffffff	32	0x00000001	31	0xffffffff	32
0xcccccccc	16	0x00000004	2	0x33333333	16
0x33333333	32	0x00000003	3	0x11111111	8
0xaaaaaaaa	16	0x22222222	9	0x00000005	9
0x55555555	32	0x55555555	24	0x00000001	1
0x80000000	17	0x08888888	23	0x0000000f	3
0xeeeeeeee	23	0x00000002	8	0x77777777	21
0x11111110	31	0x00000002	0	0x08888888	31
0xcccccccc	23	0x00000003	1	0x44444444	15
0xffffffff	16	0x0000ffff	14	0x00010001	10
0xffff0000	16	0x0000ffff	0	0x00010000	1
0x0000ffff	32	0x0000000f	12	0x00001111	5
0xffffffff	16	0xffff0000	20	0x00000001	4
0x00000001	31	0xffffffff	16	0xffffffff	31
0x44444444	9	0x00000002	31	0x22222222	24

As in the multiplication routine, some dependent divisions are performed where each subsequent divide depends on the previous quotient and a final check is performed at the end. Table 5 shows the computations for each step, each intermediate value, the final result expected, and the number of bits switching in each intermediate result. While this does not accomplish perfect switching among the bits for each intermediate result, it is satisfactory in attempting to catch control flow errors in the division loop.

Table 5: Dependent Integer Division Vectors.

Divide Operation	Intermediate result	Bits Switching
`UiR0 = 0xf0000000;`	`0xf0000000`	
`uiR0 = uiR0 / 0x00000001;`	`0xf0000000`	0
`uiR0 = uiR0 / 0x0000000f;`	`0x10000000`	3
`uiR0 = uiR0 / 0x0000000f;`	`0x01111111`	8
`uiR0 = uiR0 / 0x0000000f;`	`0x00123456`	9
`uiR0 = uiR0 / 0x0000000f;`	`0x000136b0`	9
`uiR0 = uiR0 / 0x0000000f;`	`0x000014b6`	5
`uiR0 = uiR0 / 0x0000000f;`	`0x00000161`	9
`uiR0 = uiR0 / 0x0000000f;`	`0x00000017`	6
`uiR0 = uiR0 / 0x0000000f;`	`0x00000001`	3
Final Result	`0x00000001`	

6.5 Logical AND

The vectors used for the logical AND operation are derived so that every input combination for each cell in the AND gate array is implemented. These vectors are easy to develop since there are no carry bits and every gate in the computation is independent of the others. The truth table for a single AND gate cell is shown in Table 6. The columns labeled 'x' and 'y' represent the inputs, and the column labeled 'x AND y' represents the output for the corresponding inputs. If there is a fault in any gate in the cell array, one of these input vectors will propagate the error to the output so that it can be detected.

Figure 6 shows an example of the code using the input vectors outlined in Table 6. The format of the code for all of the integer routines is essentially the same. The operations are performed with the defined vectors and then subsequent consistency checks are executed to check for errors. This process is then repeated in a loop.

Table 6: Truth Table for AND Function.

X	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

In addition to the input vectors described in Table 6, other random vectors are used to increase the number of logical AND operations to be executed. It is important to flood the hardware with enough operations so that it is constantly in use and so that the error latency is minimized.

```
CheckLogicalAND()

 register int R0, R1, R2, R3;

 for (int i=0; i < LOOP_ITERATIONS; i++) {
  R0 = 0x00000000 & 0x00000000; // inputs 00
  R1 = 0x00000000 & 0xffffffff; // inputs 01
  R2 = 0xffffffff & 0x00000000; // inputs 10
  R3 = 0xffffffff & 0xffffffff; // inputs 11

  if (R0 != 0x00000000) UpdateStats(Err, la);
  if (R1 != 0x00000000) UpdateStats(Err, la);
  if (R2 != 0x00000000) UpdateStats(Err, la);
  if (R3 != 0xffffffff) UpdateStats(Err, la);
 }
```

Figure 6: Example Code for the Logical AND Routine.

6.6 Logical OR

The routine to test the logical OR hardware is similar to that of the logical AND. Table 7 shows the truth table for the logical OR function. As in the logical AND routine, all of the possible inputs combinations into the logical OR hardware are tested. Figure 7 exemplifies the code used in testing the logical OR routine. Also, as in the logical AND routine, other input vectors are used to flood the hardware with computations, each of which aims to maximize switching in the hardware.

Table 7: Truth Table for OR Function.

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

```
CheckLogicalOR()

 register int R0, R1, R2, R3;

 for (int i=0; i < LOOP_ITERATIONS; i++) {
   R0 = 0x00000000 | 0x00000000; // inputs 00
   R1 = 0x00000000 | 0xffffffff; // inputs 01
   R2 = 0xffffffff | 0x00000000; // inputs 10
   R3 = 0xffffffff | 0xffffffff; // inputs 11

   if (R0 != 0x00000000) UpdateStats(Err, lo);
   if (R1 != 0xffffffff) UpdateStats(Err, lo);
   if (R2 != 0xffffffff) UpdateStats(Err, lo);
   if (R3 != 0xffffffff) UpdateStats(Err, lo);
 }
```

Figure 7: Example Code for the Logical OR Routine.

6.7 Logical XOR

The logical XOR routine follows the same ideas as that of the logical AND and the logical OR routines. The truth table for the logical XOR operation is shown in Table 8, along with a code excerpt in Figure 8. Again, other vectors are also used to flood the hardware with computations.

Table 8: Truth Table for XOR Function.

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

```
CheckLogicalXOR()

 register int R0, R1, R2, R3;

 for (int i=0; i < LOOP_ITERATIONS; i++) {
   R0 = 0x00000000 ^ 0x00000000; // inputs 00
   R1 = 0x00000000 ^ 0xffffffff; // inputs 01
   R2 = 0xffffffff ^ 0x00000000; // inputs 10
   R3 = 0xffffffff ^ 0xffffffff; // inputs 11

   if (R0 != 0x00000000) UpdateStats(Err, lx);
   if (R1 != 0xffffffff) UpdateStats(Err, lx);
   if (R2 != 0xffffffff) UpdateStats(Err, lx);
   if (R3 != 0x00000000) UpdateStats(Err, lx);
 }
```

Figure 8: Example Code for the Logical XOR Routine.

7 TESTING THE REGISTER UNIT

The register unit should be one of the units tested the most. If there are errors present in the register unit, then the entire microprocessor will produce faulty results. The processor is assumed to have 32 general purpose registers (GPRs), each 32 bits wide.

7.1 Fault Model

Errors occurring in the register unit are classified in one of the following ways:

1. No register is accessed or a nonexistent register is accessed.
2. An incorrect register is accessed.
3. One or more of the bits in the register data is wrong.

By moving specific data to and from registers in the C program, checking the values, and then repeating, the above types of errors can be detected. However, the test does not guarantee that all registers will be utilized as desired. A C program can only recommend that a register be used for a variable. This is achieved with the declaration of `register` before the type of variable, for example, `register int x`. The compiler will do its best to use a register for the particular variable.

7.2 Functional Vectors

The functional vectors for the register unit are simple to derive. By changing the value of a register to a value that is exactly opposite of its previous value, the switching in the bits will be maximized and there will be a higher likelihood of detecting an error. For example, if we load a register with 0x33333333, check the result, then load the register with 0xcccccccc and check the result again, we accomplish the high level of switching in the bits. Also, it is important to use unique data values for each register when testing them so that if the wrong register is chosen, as in condition 2 above, the consistency check does not coincidentally get the correct value. Figure 9 shows an example of a portion of the code.

```
CheckRegisterUnit()

 register int R0, R1, R2;

 for (int i=0; i < LOOP_ITERATIONS; i++) {
   R0 = 0x00000000;
   R1 = 0x11111111;
   R2 = 0x22222222;

   if (R0 != 0x00000000) UpdateStats(1, ru);
   if (R1 != 0x11111111) UpdateStats(1, ru);
   if (R2 != 0x22222222) UpdateStats(1, ru);

   R0 = 0xffffffff;  // exact opposite
   R1 = 0xeeeeeeee;  // of previous values.
   R2 = 0xdddddddd;

   if (R0 != 0xffffffff) UpdateStats(1, ru);
   if (R1 != 0xeeeeeeee) UpdateStats(1, ru);
   if (R2 != 0xdddddddd) UpdateStats(1, ru);
 }
```

Figure 9: Example Code for the Register Unit.

8 TESTING THE LOAD/STORE UNIT

The purpose of the load/store routine is to test the hardware associated with loads and stores. This includes the hardware for the effective address calculation, any load or store queues, and any other related hardware involved in the load/store unit. Note that this routine is not intended to explicitly test the memory or the data cache. A separate routine is used to test the data cache and is explained in the thesis to be written by Hari Kommaraju [18].

8.1 Fault Model

The fault model for a load or store operation assumes the following types of errors:

1. An error in the memory address of the load or store. This can be due to a bit flip in the address or to an incorrect calculation of the effective address.
2. An error in the data for the load or store.

8.2 Functional Vectors

The functional vectors for the load/store unit attempt to catch the errors described in the fault model above. Arrays are used to allocate areas in memory and transfer data to and from these locations. The data that is transferred is chosen in such a way as to maximize the switching of the bit values in each memory location. The following paragraph outlines this process.

First, ten integer arrays with sixteen locations each are allocated and initialized to specific values, for example, `int a[16] = {0xaaaaaaaa, 0xaaaaaaaa, … }`. Ten arrays are chosen so that a reasonable amount of memory is allocated each time the routine is called. Second, the data from one initialized array is transferred to the corresponding location of another array. This operation consists of loading the value from one location and storing it to another. Third, the values in each location of the array with new data are checked for the correct value. This involves an additional load operation. Finally, some dependent load and stores are executed to prevent any compiler optimizations. Figure 10 shows an example of the code.

By transferring the data from one array to another in this manner, any bits that have been erroneously flipped in the data will be detected in the consistency checks. Similarly, any bits in the address of the memory location that are changed or erroneously calculated will result in the wrong memory location being accessed and will also be detected in the consistency check.

```
CheckLoadStoreUnit()
   int g[16] = {0x33333333, 0x33333333, …, 0x33333333};
   int h[16] = {0xcccccccc, 0xcccccccc, …, 0xcccccccc};

 for (int i=0; i < LOOP_ITERATIONS; i++) {
   h[0] = g[0];
   h[1] = g[1]; . . .
   h[15] = g[15];

   if (h[0] != 0x33333333) UpdateStats(Err, lsu);
   if (h[1] != 0x33333333) UpdateStats(Err, lsu); . . .
   if (h[15] != 0x33333333) UpdateStats(Err, lsu);

   // Dependent loads and stores
   g[0] = 0xcccccccc;  // Opposite of previous value
   g[1] = g[0]; . . .
   g[15] = g[14];

   if (g[0] != 0xcccccccc) UpdateStats(Err, lsu);
   if (g[1] != 0xcccccccc) UpdateStats(Err, lsu); . . .
   if (g[15] != 0xcccccccc) UpdateStats(Err, lsu);

   // Copy back the contents for the next loop
   g[0] = h[0];
   g[1] = h[1];
   g[15] = h[15];
 }
```

Figure 10: Example Code for the Load/Store Unit.

9 TESTING THE INSTRUCTION FETCH UNIT

The instruction fetch unit encompasses all of the hardware involved in the instruction fetch cycle. This typically involves hardware such as the instruction register, an instruction cache, and perhaps an instruction queue. There may be other hardware as well. The general idea of testing the instruction fetch unit is by means of executing instructions. The instruction cache behaves as any cache, but cannot contain just any data. It must contain valid instructions. Potential cache faults result in an alteration of these instructions that may lead to errors in the program.

9.1 Fault Model

An operation in a high-level program can be viewed as a sequence of one or more machine-level instructions, where every instruction represents the elementary data-transfer and data-manipulation operations for a given architecture. Figure 11 shows an example of some C program statements that are converted to possible machine-level instructions.

C Code	Possible Corresponding Machine-level instructions
`int R1 = 4;`	`mov R1, 4`
`int R2 = 8;`	`mov R2, 8`
`int R3 = 55;`	`mov R3, 55`
`R0 = R1 + R2 - R3;`	`add R0, R1, R2`
	`sub R0, R0, R3`
`if (R0 != 12) Err++;`	`beq R0, 12, #done`
	`addi R0, R0, 1`
	`done:`

Figure 11: Example Machine-Level Instructions.

The general form of an instruction is shown in Figure 12, where the opcode is the operation code for the given instruction and A, B, and C are the addressing fields. The addressing fields can be one of the following:

- a register number,
- a constant value,
- or an offset.

opcode	A	B	C

Figure 12: General Instruction Format.

Errors in the instruction fetch unit are related to the machine-level instructions and will occur under one or more of the following events:

1. One or more instructions become inactive; therefore, the operation is not executed completely.
2. Instructions that are normally inactive become active. This includes an instruction opcode that gets changed in such a way so that another instruction gets activated. For example, if the opcode for the ADD instruction is 0001 and the opcode for the OR instruction is 0011, it is conceivable that the ADD instruction opcode may get changed to the OR instruction in the event of a SEU.
3. One or more of the addressing fields of the instruction is changed to the wrong value.
4. A branch to an incorrect address somewhere within the software tool that was not intended.

In the first case, if the operation is not executed completely, then the final result will be incorrect and the check instruction will catch the error. In the second case, either a wrong, but valid instruction will get executed, or an invalid instruction will get executed. If a valid, but wrong instruction gets executed, the check instruction should catch the error unless the wrong instruction is a branch. If that is true, as long as the branch jumps to a location within the program space as in the fourth case, the program is capable of reporting the error. However, if the branch jumps outside of the program space, an exception will occur and the program will crash. If an invalid instruction gets executed, an illegal instruction exception will occur and also cause the system to crash. If an error occurs in one of the addressing fields, several scenarios are possible:

1. The wrong register is used.
2. An invalid register is used.
3. An error in the constant value is used.
4. An invalid constant is used.
5. A wrong offset value is used.
6. An invalid offset is used.

In 1 and 3, the error will be detected by the check instruction. In 2, 4, and 6, the system will most likely generate an exception and the system will crash. In the case of 5, if the error in the offset causes the instruction to jump to another location in the program, the error will most likely be detected. If the error causes the program to jump to a location outside of the program space, the system will generate a segmentation fault and crash.

In general, any instructions that are affected in such a way so that an illegal instruction is generated, or that a branch to an illegal portion of memory is executed, the operating system will produce a segmentation fault and crash the program. These types of errors are highly likely and there is nothing that can be done when they occur. It is the less severe, wrong but still legal instructions that our program hopes to detect and report.

The above fault model is fairly straightforward, but it is still difficult to explicitly test the instruction fetch unit and even more tricky to attribute errors to its hardware. The general idea behind the test routine in this program is to execute several types of operations and monitor the types of errors found. For example, if there are errors occurring in many different places in the instruction fetch routine (i.e., errors in more than one type of operation), we can make the hypothesis that there are upsets somewhere in the instruction fetch unit. This correlates with the assumptions that any SEUs occurring will be localized to a specific area and are fairly infrequent.

In addition, by comparing the output from the routine for this unit with the output from other units, it is possible to further conclude that certain types of errors are most likely occurring in the instruction fetching hardware. As an example, if the integer operations or the floating point operations do not produce any errors when running their individual routines, but they all produce errors when running the instruction fetch routine, then there is most likely a problem in the instruction fetching hardware.

9.2 Functional Vectors

As stated in the previous section, the test code for this unit executes various instructions of different types. Each instruction performs a data manipulation or a data transfer, and at the completion of the instruction, the expected value is checked. If there is an error in the instruction performing the operation to be checked such that the processor is able to continue executing, it should be caught by the following consistency check.

A portion of the code used to check the instruction fetch unit is shown in Figure 13. The actual routine has more operations included. It can be seen that there are many types of computations performed and that unless there are multiple types of errors found, it cannot be assumed that the errors are occurring in the instruction fetch unit. In that case, further tests must be run. But, if there are multiple errors detected in this routine, then the situation is reported and the appropriate action can be taken.

```
void CheckInstructionFetchUnit()

   register long   iR0, iR1, iR2, iR3, iR4, iR5, iR6, iR7, iR8;
   register double fR0, fR1, fR2, fR3, fR4, fR5, fR6, fR7, fR8;
   long   i[4] = {12, 3, 192, 48};
   double m[4] = {12001.2, 3000.3, 192019.2, 48004.8};

  for (int i=0; i < LOOP_ITERATIONS; i++) {
   iR0 = i[0];            // load
   if (iR0 !=  12) { ErrType = 1; goto END; }
   iR4 = iR0 + iR1;       // integer add
   if (iR4 !=  15) { ErrType = 2; goto END; }
   fR0 = m[0];            // load
   if ((int)(fR0*10) != 120012) { ErrType = 1; goto END; }
   fR4 = fR0 + fR1;       // floating point add
   if (fR4 !=  15001.5) { ErrType = 6; goto END; }
   iR4 = iR6 - iR4;       // integer sub
   if (iR4 != 240) { ErrType = 3; goto END; }
   fR4 = fR6 - fR4;       // floating point subtract
   if (fR4 != 240024.0) { ErrType = 7; goto END; }
   iR6 = iR4 * iR5;       // integer multiply
   if (iR6 !=      3600) { ErrType = 4; goto END; }
   fR6 = fR4 * fR5;       // floating point multiply
   if ( fR6 != 3600720036.0 ) { ErrType = 8; goto END; }
   iR7 = iR7 / 10000;     // floating point divide
   if (iR7 != 4095) { ErrType = 5; goto END; }
   iR8 = iR7 / iR5;       // integer divide
   if (iR8 != 273) { ErrType = 5; goto END; }

END: if (ErrType != PrevErrType) {
      printf("Err in IFU: Multiple Instructions have errors.\n");
      UpdateStats(1, IFUnit);
      PrevErrType = ErrType;
   } else SameErrs++;
  }
```

Figure 13: Example Code for the Intruction Fetch Unit.

10 TESTING THE BRANCH PROCESSING UNIT

The branch processing unit includes any hardware used in branch prediction or branch control, for example the branch target instruction cache, the branch history table, and perhaps some control registers. The routine to test the branch processing unit aims at detecting any control flow errors occurring as a result of corrupted branches or branch addresses.

10.1 Fault Model

Errors expected in the branch processing unit are as follows:

1. The target address for the branch is wrong
2. One or more errors in the control registers associated with branch processing
3. One or more errors in the status bits

Errors in the actual prediction for a given branch cannot be detected from a high level program. If a SEU affects the prediction for a certain branch, no error will be produced. The worst that will happen is the processor will have to backtrack and re-execute the branch with the correct path.

10.2 Functional Vectors

The branch processing unit is tested using an `if-then-else` tree. A pictorial representation of the `if-then-else` tree is shown in Figure 14. Each circle in the tree represents an `if` statement. If the condition of the `if` statement evaluates to true, then the right branch of the tree is followed; otherwise the left branch is followed—i.e., `if` condition = TRUE `then` go right, `else` go left. At the end of each path resides a leaf node containing a numerical value called the condition code. Each condition code is unique and represents a specific path in the tree.

To use this tree structure in testing the branch processing unit, we begin by defining a global variable to be set to the value of the condition code of the expected path to be taken in the tree. We will call this variable `ConditionCode`. Next, we represent the left branch of every node in the tree with a 0, or FALSE value, and the right branch of every node with a 1, or TRUE value. Also, we define a set of bits—one for each level in the tree—to be used as the condition for each `if` statement. The `ConditionCode` variable is set to correspond to value of the condition bits. For example, a `ConditionCode` value of 10 will correspond to the condition bits—starting with level one—1010 (right, left, right, left).

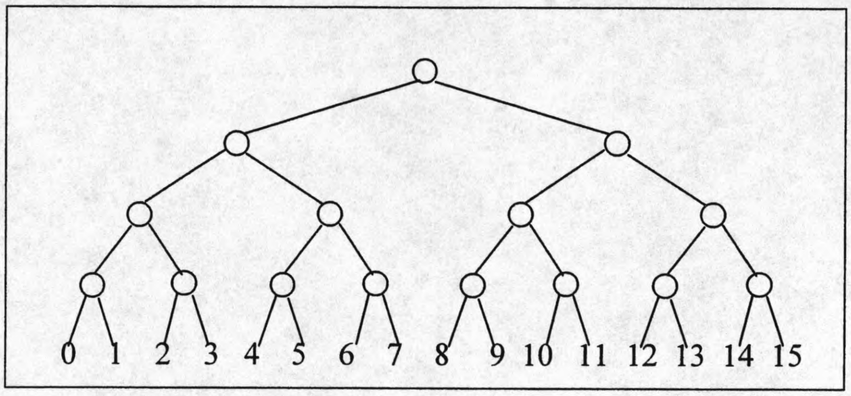

Figure 14: If-Then-Else Tree for the Branch Prediction Unit.

As stated above, if the bit for a particular level in the tree is 0, or FALSE, then the left branch will be taken; otherwise, the bit is a 1, or TRUE, and the right branch will be taken. In this way, we can control the flow of the program. We check for errors when we reach a leaf node. If the `ConditionCode` variable is set to what we would expect it to be at this node, then no error exists. However, if the `ConditionCode` variable is set to something different than expected, an error has occurred. In addition to checking for the expected value of the `ConditionCode` variable at each node, we also set the condition bits for each `if` statement to new values and update the `ConditionCode` variable so that a different path in the tree is executed the next time through. Once all the paths in the tree have been taken, the process repeats. Figure 15 shows a code example using a two-level tree with the above strategies.

The actual routine for this unit uses a tree with six levels. The idea is to create a big enough tree with enough branches so that if a control-flow error occurs and the branch address is corrupted, there will be a reasonable probability that it will branch somewhere else in the tree and the error will be detected because the value of the `ConditionCode` variable will be wrong. Also, if the status bits or the control registers are corrupted, then a branch may not follow the appropriate path, and again the error will be detected.

```
int ConditionCode;

void CheckBranchProcessingUnit() {

  register unsigned short b0 = 1, b1 = 1;
  ConditionCode=3;

  for (done = 0; done < LOOP_ITERATIONS; done++) {
     if (b0) {
      if (b1) {
        if (ConditionCode != 3) { UpdateStats(1, BPUnit); }
        else {  ConditionCode = 2;
           b0 = 1;
           b1 = 0;
        }
      else {
         if (ConditionCode != 2) { UpdateStats(1, BPUnit); }
         else { ConditionCode = 1;
            b0 = 0;
            b1 = 1;
    else {
      if (b1) {
        if (ConditionCode != 1) { UpdateStats(1, BPUnit); }
        else {  ConditionCode = 0;
           b0 = 0;
           b1 = 0;
        }
      else {
         if (ConditionCode != 0) { UpdateStats(1, BPUnit); }
         else { ConditionCode = 3;
            b0 = 1;
            b1 = 1;
         }
      }
    } if (b0)
  } // for loop
```

Figure 15: Example Code for the Branch Processing Unit.

11 FAULT INJECTION METHODS

Fault insertion, the deliberate injection of faults into a system, is a useful validation method in any system development process where the error handling and detection is a vital concept. Typically, fault insertion techniques fall into two main categories: physical, or hardware fault insertion, and simulated, or software fault insertion. For this tool, only physical fault insertion techniques are used because these are most representative of the types of errors that the tool is trying to detect. Also, in using simulation-based fault insertion techniques, a software model of the system is required which is difficult and expensive to obtain.

The most commonly used physical fault injector is heavy-ion radiation [20], but in some cases electromagnetic fields or power supply disturbances are also used. For this thesis, we present results for power supply disturbances only. Radiation experiments and other tests are scheduled at JPL in the near future. In addition, there are no known effective methods of using electromagnetic fields to produce transient errors inside the microprocessor. As a result, power supply disturbances are our best method for producing errors inside the processor so that we may test our software tool.

11.1 Radiation Injection

Bombarding integrated circuits with heavy-ion radiation or a high-energy proton beam can cause SEUs in the circuits of the processor. Irradiation of a circuit must be performed in a vacuum. This is due to the fact that air molecules attenuate heavy ions. Also, the packaging of the circuit has the potential to prevent the heavy ions from reaching the depletion regions. Such things as heat sinks and cooling towers will get in the way of radiation insertion.

11.2 Power Supply Disturbances

Disturbing the power supply is a simple way to cause errors in the system. This actually models some of the errors happening on Earth due to power surges and disturbances common in industrial applications. This method was used in [3] and [21] in conjunction with heavy-ion radiation on a MC6809E processor. Short voltage drops were caused at the power supply pin of the CPU using a MOS power transistor. A test CPU and a reference CPU were run in lock step and the external buses were compared while the power supply pin of the test CPU was being disturbed. For this project, we do not have visibility of the external buses on our test boards and cannot compare the values of a reference CPU to a test CPU. However, we accomplish the power supply disturbances by connecting a variable, digital power supply source to the power supply pin of the microprocessor. On our test boards, we are able to access the pin that leads directly to the CPU and only the CPU. Chapter 12 explains this setup in further detail.

Although the reduction in power supply voltage reduces power consumption, the real trade-off is the increase of delay. If the power supply voltage is scaled down while all other parameters are kept constant, the propagation delay time will increase. This, in turn, will affect the critical path in the system causing it to take longer to execute. If the critical path does not complete in the time allotted, an error is likely to occur. This is precisely the behavior we are hoping for. Results are presented in the next section for these tests.

12 EXPERIMENTAL RESULTS

Preliminary results on the validity of our tool have been obtained from one of the fault injection methods described in the previous section, namely power supply disturbances. As explained, power supply disturbances are currently the only available method we have at our facility to test our tool. However, further experiments using alpha-particle radiation are scheduled to take place at JPL in the future. Disturbances in the power supply are used as a way to cause errors in the microprocessor that may be detected by our software program.

When doing fault injection using radiation or power supply disturbances, we do not know where the fault is going to happen. Therefore, all parts of the chips should be exercised by using all the functional units, e.g., floating-point as well as integer units. If we fail to do this, a fault may happen somewhere and never show up as an error. If we exercise all units and output the results, we can detect errors happening almost anywhere in the chip.

12.1 Experimental Setup

As explained in Chapter 5, the PowerPC MPC750 is the processor for which the preliminary version of our tool is based. The test boards containing the MPC750 processor are based on an embedded system and are referred to as single board computers (SBCs). In our case, the board is referred to as the SBC750.

The experimental setup for our tests includes several parts:

- the target system under test – WindRiver's SBC750,
- a Hewlett Packard E3631A Power Supply,
- a host computer,
- debugging software, and
- HyperTerminal software used to obtain the serial communication between the host computer and the target.

Figure 16 shows an example of the hardware setup used for these experiments.

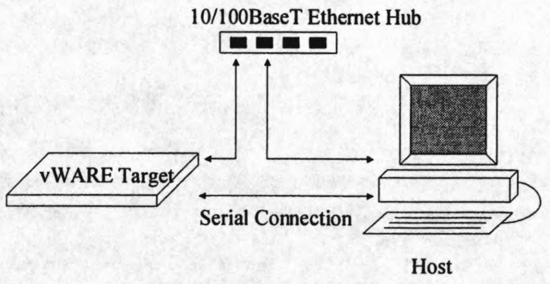

Figure 16: Sample Hardware Setup.

Once the hardware setup is ready, our tool is downloaded to the target board as part of the firmware. Then, from a HyperTerminal window on the host computer, the program is run and the output is monitored. Any occurrences of errors are reported through the use of print statements to the HyperTerminal window. An EEPROM is included on the board and some of the statistics are also written to this memory. The EEPROM is nonvolatile and can only be erased if explicitly told to do so. Therefore, any data written to this area does not get lost when the board is reset. This is useful when the program hangs, or if an error occurs while printing to the HyperTerminal window.

The firmware that is loaded onto the board is another product of WindRiver. It is called visionWARE and provides the basic communication with the target along with components such as device drivers, diagnostics, flash programming, and network support. A user's manual for setting up the hardware and running the tool is included in Appendix A.

12.2 Program Output

The program is designed to output statements that give helpful information about the state of the system. If an error is detected, the program will print the error, it will print which routine detected the error, and it will print some statistical information. All of the statistical information is calculated in one routine called `UpdateStats()`. The `UpdateStats()` routine is called every time an error is detected, as well as when each routine completes. It keeps track of how long each routine has been executing along with some basic statistics. If an error is found, the information printed includes

- the line of code corresponding to the consistency check that failed,
- the total number of errors found in the particular routine,
- the error rate of the particular routine,
- the mean time between errors (MTBE), and
- the total number of errors found up to that point.

The error rate is calculated by dividing the number of errors by the total running time of the routine up to the time of the error. The MTBE is calculated by dividing the running time by the number of errors. It is difficult to be more precise or to add additional calculations without significantly increasing the overhead of the program. The next section discusses the overhead in more detail.

12.3 Overhead

The overhead in this program includes the code to start up the program, to check for errors, to calculate the statistics, to print the information to the screen, and for basic flow control. Over half of each routine includes instructions that are involved in the consistency checks, the print statements, and the flow control. This leaves a reasonably high level of probability that errors will occur in one of the overhead instructions instead of one of the test instructions. However, even if errors occur in these other statistical-type instructions, it is still possible for the program to detect that there is a problem.

12.4 Programming Issues

One of the main programming issues faced when attempting to test our program is the problem of compiler optimizations. Most compilers offer high levels of optimizations that intend to improve the code and make it faster to execute on the processor. In our case, however, we do not desire such optimizations. “Smart” compilers may regard many of our computations as unnecessary. These compilers have only the final result in mind and will nullify our attempts at testing specific hardware for errors. For example, the compilers might change a conditional branch to an unconditional branch if the condition is a constant value.

Another issue is the amount of CPU time that the program uses. If transient errors are present, our diagnostic program requires the majority of the CPU time when executing.

Otherwise, any applications running at the same time that require intensive CPU usage will likely be the main culprit for any errors. The operating system alone requires a certain amount of CPU time that is unavoidable, but exactly how much CPU time will vary. We ran an experiment to determine the approximate amount of CPU time our program would obtain when no other applications other than the operating system were running. The results are presented in the next section.

A third issue is to reduce the error latency as much as possible by flooding the execution units with millions of operations. Once a fault occurs, it may take several clock cycles before it propagates to an observable error. By maximizing the number of computations performed in each unit, this error latency is minimized. Table 9 outlines the number of computations that are performed in each routine along with the approximate number of operations executed every second.

The speed of the MPC750 is 233 MHz, so in the best case, we can expect $233\text{x}10^6$ operations per second. That is, one operation occurring per clock cycle. In that case, if a transient error were to occur, it will be detected on the very next clock cycle. From the table, we can see that the logical operations are approximately equal to this best-case scenario. Keeping in mind that the running time and the operations per second in the table are approximate, it should also be noted that logical operations are performed two at a time, one in each integer unit. Therefore, the best-case scenario for the logical operations is actually twice the speed of the processor—466 MHz. The error latency for the logical routines is then estimated as two clock cycles. The error latency for other routines can be estimated in the same way.

The overhead involved in each routine should also be taken into account when considering the number of operations per second. For every operation used to test a particular unit, there is a consistency check involved. The consistency check operations are not included in the count for the total operations in Table 9, but the time that they take to execute is included in the approximate running time.

Another issue involving overhead includes operating system requirements. In our test environment, the board we used did not have an operating system running on it. In commercial systems, however, there will be an operating system that will require some amount of CPU time. To gauge approximately how much CPU time a typical operating system needs, we ran our program on an HP PA-RISC Unix machine while no other programs were running. Using the

Unix-based time command, we found that our program will get approximately 80% of the CPU cycles, and the operating system requires about 20% of the CPU cycles. The percentages were produced based on the program's total CPU time, as a percentage of elapsed time.

Table 9: Operations and Running Times for Each Routine.

Routine	Operations per loop	Number of loops	Total Operations	Approx. Running Time	Approx. Operations Per Second
Register Unit	40	8,000,000	320,000,000	6.34 s	50.47×10^6
Instruction Fetch Unit	32	8,000,000	256,000,000	92.04 s	2.78×10^6
Integer Addition	40	8,000,000	320,000,000	9.35 s	34.22×10^6
Integer Subtraction	40	8,000,000	320,000,000	9.12 s	35.09×10^6
Integer Multiplication	58	8,000,000	464,000,000	18.21 s	25.48×10^6
Integer Division	50	8,000,000	400,000,000	33.72 s	11.86×10^6
Logical AND	20	8,000,000	160,000,000	0.71 s	225.35×10^6
Logical OR	20	8,000,000	160,000,000	0.64 s	250.00×10^6
Logical XOR	20	8,000,000	160,000,000	0.71 s	225.35×10^6
Integer Unit 2	40 adds & multiplies	8,000,000	640,000,000	48.75 s	13.13×10^6
Floating Point Add	20	8,000,000	160,000,000	0.82 s	195.12×10^6
Floating Point Subtract	20	8,000,000	160,000,000	0.82 s	195.12×10^6
Floating Point Multiply	20	8,000,000	160,000,000	0.83 s	192.77×10^6
Floating Point Divide	20	8,000,000	160,000,000	0.82 s	195.12×10^6
Branch Processing Unit	7	8,000,000	56,000,000	6.09 s	9.20×10^6
Load/Store Unit	320 loads, 192 stores	80,000	40,960,000	13.24 s	3.09×10^6
Data Cache	2	3,300,000	6,600,000	15.97 s	0.41×10^6

12.5 Results

This section shows the results obtained from various experiments. The hardware test setup and some of the issues involved have been discussed in the previous sections. The results presented here include those for the following experiments:

- power supply disturbances,
- weak radiation exposure, and
- temperature increases coupled with power supply disturbances.

12.5.1 Power supply disturbances

The most interesting results obtained were those from the power supply disturbances. Four different boards were used to verify consistent behavior on identical platforms. The power supply disturbances were generated by attaching the HP digital power supply unit to the power supply pin of the processor and then reducing the voltage. Table 10 gives a summary of the data gathered. The purpose of these tests was to determine if our software would be able to detect errors occurring in the microprocessor. The goal was not to obtain an accurate number of how many power supply disturbances will result in hangs or how many errors will be detected. We simply wanted to verify program functionality.

In the table, the "No. Tests" column enumerates the number of tests resulting in a problem with the program, whether it was a hang, or some kind of error. The "Voltage Range" is the range of voltages used to produce the errors. The lower voltage in the voltage range represents the threshold voltage at which the program could not operate at all. The upper voltage in the range represents the lowest value that the voltage could be reduced to without causing the program to result in an error or hang. The "Hangs" column represents the number of tests from column 2 that resulted in a hang. Similarly, the "Program Detected" column represents the number of tests from column 2 that resulted in an error detected by the program.

Table 10: Results from the Power Supply Disturbance Experiments.

Board No.	No. Tests	Voltage Range (in Volts)	Control Flow Errors Detected	
			Hangs	Program Detected
1	45	1.99 – 2.10	31	14
2	35	2.00 – 2.08	26	9
3	25	2.10 – 2.29	18	7
4	25	2.08 – 2.20	17	8

We expect that the errors produced in the microprocessor will fall under one of two main categories: control flow errors and data errors. Control flow errors (sequencing errors) are associated with upsets in the program counter (PC), certain special function registers (e.g. control registers), and status registers. Data errors, on the other hand, are associated with upsets in the data memory, counters, etc. Control flow errors are generally more severe than data errors, which simply change data words.

The types of errors detected by our program due to the power supply disturbances appeared to be control flow errors. There were no data errors found. Either the program would hang and no error statements were printed, or error statements were printed and the program would eventually hang. The latter were classified as program detected control flow errors. Sometimes, the errors detected by the program were caught by the branch processing unit, and other times the print statements would output at the wrong times, indicating erroneous jumps in the program. Also, when the power supply voltages were dropped and held constant, multiple runs at the same voltage resulted in similar outputs. In those cases, the program would usually fail in the same location and produce the same type of errors. Power supply drops are modeled by delay faults and delay faults will affect the critical path in the system. If the exact same program is run at the same voltage, then the same critical path will usually be affected and the program will fail in the same place every time.

12.5.2 Weak radiation exposure

A third experiment involved exposing the microprocessor to a weak radiation source emanating from an ionization chamber out of a smoke detector. Inside the ionization chamber is a small amount of Americium-241 (perhaps 0.0002 g). The radioactive element Americium has a half-life of 432 years, and is a good source of alpha particles. Although the amount of radiation in a smoke detector is predominantly alpha radiation, it is of extremely small quantity. Also, almost all of the alpha radiation particles cannot penetrate a sheet of paper, and it is blocked by several centimeters of air. This is why a special facility is needed to do the radiation testing that involves a vacuum and a particle beam.

The ionization chamber was removed from the smoke detector and placed directly on top of the microprocessor. The microprocessor has a cover over it and because most alpha particles cannot penetrate paper, they also cannot penetrate a metal cover. However, the motivation behind this experiment was to see if enough particles actually did get past the cover of the processor and into the circuitry to cause an error. The laws of probability suggest that it is possible for a few of the particles to penetrate the metal cover.

If an error were to be detected by our tool during the time that the ionization chamber is placed on the microprocessor, we would be able to express confidence that our program is working as intended. However, the program was run for several days with the Americium-241 radiation in place and no errors were found. This does not prove anything negative about our program; it simply tells us that further tests are required. It does provide an interesting experiment, however.

12.5.3 Temperature increases coupled with power supply disturbances

The last type of experiments involved raising the temperature of the processor environment while at the same time lowering the power supply voltage. The idea here was to run our tool at the lowest voltage possible without producing any errors or hangs, and then increase the temperature in hopes of generating different types of errors than those of dropping the power supply voltage alone. Significant increases in temperature will reduce the performance of the processor causing the gates to switch at a slower speed. This is similar to the delay faults caused by power supply drops. Unfortunately, our methods of temperature increase were informal and unregulated and did not produce any significant results. However, the idea may be useful and provide supportive data if more official laboratory setups were available to perform the experiments.

13 CONCLUSIONS AND FUTURE WORK

The goal of the NASA HPCC Remote Exploration and Experimentation (REE) project is to transfer commercial supercomputing technology into space. The difficulty NASA is encountering is that radiation hardened components are both extremely expensive and lag several generations behind the commercial state of the art components.

The goal of this project is to provide a tool to help evaluate and analyze the vulnerability of microprocessors exposed to alpha-particle radiation in space applications. Its intention is to provide information on the current state of the system and collect as many in-flight transient errors as possible to try and prevent any catastrophic events. By identifying and localizing errors occurring in the processor, appropriate action can be taken to prevent or alleviate further errors from occurring before the system as a whole crashes or becomes unstable.

We have shown many advantages of using this type of self-monitoring software tool for error measurements. It has excellent potential for accurate localization of a single-event upset (SEU) to within a small functional block. The error latency can be brought down to one to a few instruction cycles in most cases. Also, the probability of capturing an SEU is very high. The tool is designed to be portable to different systems as well as to be used during long space missions to periodically measure the error rates. Such a tool can give an early warning to the system of increased error rates.

We have presented results from power-supply disturbances as a method for physical-fault injection in the evaluation of the software tool. The results show that our tool is capable of detecting errors within the microprocessor as well as successfully reporting those errors. Although the tool is currently in a preliminary state, it has excellent potential to provide the type of reliability and error detection that the REE project is looking for. However, there are still areas of possible improvement and testing to be done before the final version is solidified.

The tool described in this paper is in a preliminary state. There is still considerable work left to do before the research is complete:

- The most important aspect yet to be done is the radiation testing. Bombarding integrated circuits with radiation will cause the SEUs in the circuits that our tool is designed to detect and locate. This type of hardware fault insertion introduces faults within the chip in several ways. The results from these tests will give us a feel for how often the system crashes and how many of the errors produced cause the system to hang. It will also give us information on how resilient and robust our program is when encountering errors.

- Another important factor includes extensive testing on other architectures. We primarily focused on the PowerPC architecture for the experimental results. Although, we did compile and run an early version of the program on the HP PA-RISC architecture as well as the x86 (AMD and Pentium) architecture to check for syntax errors and compilation issues. But, further testing is needed to ensure that the routines will catch a significant portion of errors on other platforms.
- Other tests that will be useful involve ground-based testing. The tool can be valuable in everyday environments where error rates may be abnormally high and a tool such as this one is needed to constantly monitor the state of a system. Power surges, internal noise sources, and capacitive and inductive crosstalk are all common causes of ground-based transient errors. The effectiveness of our software tool in these situations would be significant in determining its range of applications.
- It may be useful to add additional routines in the program to further test different parts of the hardware. Some other units that were not explicitly tested in our program may require further investigation into the feasibility of testing them from a high-level program.

REFERENCES

[1] J. R. Kimbrough et al., "Single event effects and performance predictions for applications of RISC processors," *IEEE Transactions on Nuclear Science*, vol. 41, pp. 2706-2714, December 1994.

[2] V. A. Asenek, C. I. Underwood, and M. K. Oldfield, "Predicting the rate and effects of single event upsets on satellite application software using a microprocessor simulator," in *2nd Round Table on Micro/Nano Technologies for Space*, ESA, ESTEC, October 1997.

[3] G. Miremadi, J. Karlsson, J. U. Gunneflo, and J. Torin, "Two software techniques for on-line error detection," in *Proceedings of the 22nd Annual International Symposium On Fault-Tolerant Computing*, pp. 328-335, July 1992.

[4] S. H. Crain, W. R. Crain, K. B. Crawford, S. J. Hansel, P. Yu, and R. Koga, "Single event effects test results for the 80C186 and 80C286 microprocessors and the SMJ320C30 and SMJ320C40 digital signal processors," in *Proceedings of the IEEE Radiation Effects Data Workshop*, 1998, pp. 51-57.

[5] D. M. Hiemstra, and A. Baril, "Single event upset characterization of the Pentium MMX and Pentium II microprocessors using proton irradiation," *IEEE Transactions on Nuclear Science*, vol. 46, no. 6, pp. 1453-1460, December 1999.

[6] A. Moran, K. LaBel, M. Gates, C. Seidleck, R. McGraw, M. Broida, J. Firer, and S. Sprehn, "Single event effect testing of the Intel 80386 family and the 80486 microprocessor," *IEEE Transactions on Nuclear Science*, vol. 43, no. 3, pp. 879-885, June 1996.

[7] V. Asenek et al., "SEU induced errors observed in microprocessor systems," *IEEE Transactions on Nuclear Science*, vol. 45, no. 6, pt. 1, pp. 2876-2883, December 1998.

[8] J. Beahan, L. Edmonds, R. D. Ferraro, A. Johnston, D. S. Katz, and R. R. Some, "Detailed radiation fault modeling of the Remote Exploration and Experimentation (REE) first generation testbed architecture," in *IEEE Aerospace Conference*, 2000, pp. 279-281.

[9] K.-H. Huan, and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. 33, no. 6, pp. 518-528, 1984.

[10] J.-Y. Jou and J. A. Abraham, "Fault-tolerant FFT networks," *IEEE Transactions on Computers*, vol. 37, no. 5, pp. 548-561, May 1988.

[11] J. Cusick et al., "SEU vulnerability of the Zilog Z-80 and NSC-800 microprocessors," *IEEE Transactions on Nuclear Science*, vol. NS-32, pp. 4206-4211, December 1985.

[12] M. Turmon, R. Granat, and D. S. Katz, "Software-implemented fault detection for high-performance space applications," in *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, 2000, pp. 107-116.

[13] P. Duba and R. K. Iyer, "Transient fault behavior in a microprocessor - A case study," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, 1998, pp. 272-276.

[14] S. Bagchi, B. Srinivasan, K. Whisnant, Z. Kalbarczyk, and R. K. Iyer, "Hierarchical error detection in a software implemented fault tolerance (SIFT) environment," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 203-224, March 2000.

[15] S. Bagchi, Z. Kalbarczyk, R. K. Iyer, and Y. Levendel, "Design and evaluation of preemptive control signature (PECOS) checking for distributed applications," submitted to *IEEE Transactions on Computers, Special Issues on Embedded Fault-tolerant Computer Systems*, 2001.

[16] S. Bagchi, Y. Liu, Z. Kalbarczyk, R. K. Iyer, Y. Levendel, and L. Votta, "A framework for database audit and control flow checking for a wireless telephone network controller," to appear in *Proc. of Conference on Dependable Systems and Networks*, DSN'01, July 2001.

[17] B. W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems.* Reading, PA: Addison-Wesley, 1989.

[18] H. Kommaraju, master's thesis in progress, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 2001.

[19] W.-T. Cheng and J.H. Patel, "A minimum test set for multiple-fault detection in ripple-carry adders," *IEEE Transactions on Computers*, vol. C-36, pp. 891-895, July 1987.

[20] J. Karlsson, U. Gunneflow, and J. Torin, "Use of heavy-ion radiation from 252Californium for fault injection experiments," in *Proceedings Of 1st IFIP Working Conference On Dependable Computing for Critical Applications*, pp. 79-84, August 1989.

[21] G. Miremadi and J. Torin, "Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection," *IEEE Transactions on Reliability*, vol. 44, no. 3, pp. 441-454, September 1995.

APPENDIX A REETOOL USER'S GUIDE

The REETool is intended to run continuously in the background of a computer system. It is currently being tested using a MPC750 PowerPC processor on a single board computer (SBC603/740/750). The intention of this tool is to be compiled and run on any processor, however this user's guide describes how to run the tool on the SBC603/740/750.

A.1 Setting Up the Board With the Serial Connection

To set up the board and run the program, the following things are needed:

- A host computer with the vDESKTOP software installed
    - The software for vDESKTOP is on the CD entitled "SINGLE BOARD COMPUTER Software & Documents"
- An available serial port
- An available Ethernet port
- The SBC750 board along with the Ethernet transceiver that comes with it

First, install the vDESKTOP software that comes on the CD entitled "SINGLE BOARD COMPUTER Software & Documents." The setup file is located in the vDESKTOP directory on the CD. Second, connect the serial port on the test board with either the COM1 or COM2 port on the host computer. Third, connect the board to the network by first attaching the Ethernet transceiver to the JP28 slot on the board and then connecting the network cable to the transceiver. Read the following sections to learn how to use vDESKTOP and to run the program.

NOTE: The Ethernet connection is needed only if file transfer is to be done. There is already a version of firmware flashed on the board containing the software tool. Only a serial connection is needed to run the tool. However, to update the firmware along with a new version of the tool, the Ethernet connection is required. See Section A.2 below.

Additional information for setting up the serial connection and using vDESKTOP can be found in the "visionWARE SBC.doc" file on the "SINGLE BOARD COMPUTER Software & Documents" CD under \hsidocuments\visionWARE.

A.2 Using vDESKTOP

Once vDESKTOP has been installed, it can be used to obtain a serial connection with the board. VDESKTOP uses tftp to transfer files between the host and the target board. The Ethernet transceiver is required by the tftp daemon. If no file transfer is to be done, then the Ethernet transceiver and network connection do not need to be connected.

There are three components in vDESKTOP: vCOM, vSHELL, and vFILE. VCOM uses a HyperTerminal to establish a serial connection with the board. Any input and output will be done in this window. VSHELL sets up the network parameters for the board. A valid Ethernet address, subnet mask, and gateway are needed for the Ethernet connection to work. The vFILE component brings up a file viewer so that files may be transferred between the host and the target.

A.2.1 Establishing a serial connection

To establish a serial connection,

- first click on the vCOM icon to bring up the HyperTerminal window,
- make sure the correct COM port is selected by right clicking in the HyperTerminal window and choosing Communications,
- make the connection by either clicking the circular arrow button, or selecting connect in the Tools drop-down menu.

When a connection is made, there will be a BKM> prompt in the vCOM window. If it does not work at first, try resetting the board, disconnecting and reconnecting in the HyperTerminal window.

A.2.2 Downloading the visionWARE file and updating the firmware

The tool is incorporated with the firmware file that was created with the visionWARE development kit. The development kit creates a binary file that can be used to update the memory containing the current firmware. This file, in general, is named update_*projectname*.bin. In this case it is named update_REETool.bin. To transfer the file to the target, make sure the target is on and bring up the vDESKTOP application.

Once a serial connection is made (see Section A.2.1), click on the vFILE button. This will start the tftp daemon and allow the transfer of files. In the file browser window, find the update_REETool.bin file on the host directory structure, right click on it, and choose download. Once the file is downloaded, you should be able to see a list of all the files located in RAM on the target to confirm that it is there. Next, go to the vCOM window and click in it next to the BKM> prompt so that you can type in this window. Type the following command to update the firmware:

```
update \\your_hostname\update_REETool.bin
```

When the firmware has been updated, there will be a floating-point exception that has occurred. This is not a problem, just reset the board and the new firmware should take effect.

A.3 Running the Program

To run the program, type 'reetool' following by any command line options. Table 11 shows the command line options available when running the REETool. For example, if you wanted to test only the instruction fetch unit and the branch processing unit, you would type 'reetool –ifu –bpu'. Or, if you wanted to run all of the routines, then type 'reetool –all'. To see a listing of the options from vDESKTOP, type 'help reetool', or type 'help' to see all available commands. It does not matter what order the command line options are typed. The order has no effect on how the program is run.

A.4 Statistical Output

Output from the program is accomplished with print statements that send the output through the serial port to the HyperTerminal window (vCOM) in vDESKTOP. If errors are found, a statement with the check that failed is printed (in verbose mode) along with the current error rate and the mean time between errors (MTBE).

Table 11: Command Line Options for REETool.

Command Line Option	Description
-all	Check all units
-ru	Check Register Unit
-ifu	Check Instruction Fetch
-ia	Check Integer addition
-is	Check Integer subtraction
-iu2	Check Integer unit 2
-la	Check logical AND
-lo	Check logical OR
-lx	Check logical XOR
-im	Check Integer multiplication
-id	Check Integer division
-bpu	Check Branch Processing Unit
-fpua	Check Floating Point Unit Addition
-fpus	Check Floating Point Unit Subtraction
-fpum	Check Floating Point Unit Multiplication
-fpud	Check Floating Point Unit Division
-fpu	Check All Floating Point Unit Operations
-lsu	Check Load/Store Unit
-dc	Check Data Cache
-alli	Check All the Integer Units (ia, is, im, id, iu2)
-alll	Check All the Logical Units (la, lo, lx)
-once	Run the routines specified only once.
-v	Verbose mode - prints out more detailed information about any errors found.

A.5 Making Changes to the Program

Changes to the REETool can easily be accomplished with the visionWARE development kit. Once this kit is installed, the project for REETool can be loaded from the Project→Open Project menu. The source code for all the available drivers and diagnostics is included in this project. The source code for REETool is located under the User Components→REETool→Source Files folder.

After the changes have been made, the project should be built by choosing the Build→Build vWARE command, or by clicking the build icon in the icon bar. Make sure that the build is a ROM version, not a RAM version. This can also be chosen in the build menu or from the icon bar. The ROM version creates the update_*projectname*.bin file that is used to update the firmware on the board. Once the new build file has been created, it can be loaded to the target as described above in Section A.2.