NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

COMPLETION AND TESTING OF A TMR COMPUTING
TESTBED AND RECOMMENDATIONS FOR A FLIGHT-
READY FOLLOW-ON DESIGN

by

Damen O. Hofheinz

December 2000

Thesis Co-Advisors:

Alan A. Ross
Herschel H. Loomis

Approved for public release; distribution is unlimited.

20010215 033

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)* Naval Postgraduate School	2. REPORT DATE December 2000	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Completion and Testing of a TMR Computing Testbed and Recommendations for a Flight-ready Follow-on Design	5. FUNDING NUMBERS
6. AUTHOR(S) Hofheinz, Damen O.	

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

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

13. ABSTRACT *(maximum 200 words)*

This thesis focuses on the completion and hardware testing of a fault tolerant computer system utilizing Triple Modular Redundancy (TMR). Due to the radiation environment in space, electronics in space applications must be designed to accommodate single event phenomena. While radiation hardened processors are available, they offer lower performance and higher cost than commercial off the shelf processors. In order to utilize non-hardened devices, a fault tolerance scheme such as TMR may be implemented to increase reliability in a radiation environment. The design that was completed in this effort is one such implementation.

The completion of the hardware design consisted of programming logic devices, implementing hardware design corrections, and the design of an overall system controller. The testing effort included basic power and ground verification checks to programming, executing, and evaluating programs in read only memory. During this phase, additional design changes were implemented to correct design flaws.

This thesis also evaluated the preliminary design changes required for a space implementation of this TMR design. This included design changes due to size, power, and weight restrictions. Additionally, a detailed analysis of component survivability was performed based on past radiation testing.

14. SUBJECT TERMS Fault Tolerant Computing, Triple Modular Redundancy (TMR), Commercial-Off-The-Shelf (COTS) Devices, Single Event Upsets (SEU)	15. NUMBER OF PAGES 180
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFI- CATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

i

THIS PAGE INTENTIONALLY LEFT BLANK

COMPLETION AND TESTING OF A TMR COMPUTING TESTBED AND RECOMMENDATIONS FOR A FLIGHT-READY FOLLOW-ON DESIGN

Damen O. Hofheinz
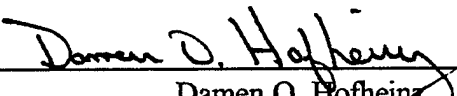Lieutenant, United States Navy
B.S., Texas A&M University, 1994

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 2000

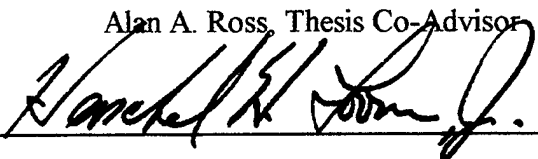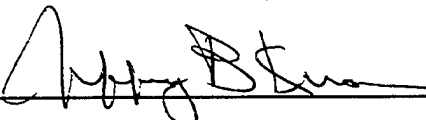Author: _____

Damen O. Hofheinz

Approved by: _____

Alan A. Ross, Thesis Co-Advisor

_____

Herschel H. Loomis, Thesis Co-Advisor

_____

Jeffrey B. Knorr, Chairman
Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis focuses on the completion and hardware testing of a fault tolerant computer system utilizing Triple Modular Redundancy (TMR). Due to the radiation environment in space, electronics in space applications must be designed to accommodate single event phenomena. While radiation hardened processors are available, they offer lower performance and higher cost than commercial off the shelf processors. In order to utilize non-hardened devices, a fault tolerance scheme such as TMR may be implemented to increase reliability in a radiation environment. The design that was completed in this effort is one such implementation.

The completion of the hardware design consisted of programming logic devices, implementing hardware design corrections, and the design of an overall system controller. The testing effort included basic power and ground verification checks to programming, executing, and evaluating programs in read only memory. During this phase, additional design changes were implemented to correct design flaws.

This thesis also evaluated the preliminary design changes required for a space implementation of this TMR design. This included design changes due to size, power, and weight restrictions. Additionally, a detailed analysis of component survivability was performed based on past radiation testing.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Increased demands on satellite performance and a declining budget have forced engineers to search for cheaper and faster products. Past satellite designs were restricted to the utilization of radiation-hardened devices, today though the use of Commercial off the Shelf (COTS) devices is increasing. In order to utilize COTS devices in space though, the engineer must employ a method of increasing the reliability and redundancy of a system such as Triple Modular Redundancy.

A Triple Modular Redundancy design was previously fabricated to prove the implementation of this concept. The objectives of this research are the completion of the design and verification of proper operation. The research began with the completion and testing of the design files previously written for the programmable logic devices utilizing WINCUPL and Xilinx design software. During this phase, a design file for a PLD was modified to correct for an error. These design files were ultimately burned into the programmable logic devices.

The last step in completing the TMR board was the design and programming of the system controller FPGA. This FPGA is responsible for data input and output, board setup, and FIFO data collection. This design was accomplished utilizing the Xilinx foundation HDL and state machine tool.

With the completion of the programming and design phases, a thorough review of the design revealed a problem with the Field Programmable Array devices. The devices utilized in the design required 3.3 Voltage, while the board was designed with a 5V bus.

The addition of a voltage regulator into the board to provide the necessary voltage for the FPGAs was the final solution to this difficulty.

Upon completion of the system design corrections, initial testing of the design for proper operation consisted of basic power and ground verification checks to executing programs in read only memory. Numerous programs were written, compiled, linked, split and burned into ROM. A digital logic analyzer was used to capture program execution to verify proper read and write cycles to RAM and ROM. The captured data provided waveforms and data lists, which confirmed correct timing. The next program transmitted serial data from the 16550 UART to a PC. Initially difficulties in obtaining output led to the discovery of an incorrect device. After obtaining the new device, the correct output baud rate and data waveforms were present on the UART. The final design program captured transmitted data from the PC, added 32, and transmitted this data to the PC.

This work also focused on the preliminary design changes necessary for space implementation of the TMR design. This included design changes due to size, power, and weight restrictions. It also included a detailed analysis of component survivability based on past radiation testing.

The effort in this work completed the design and programming of the TMR logic devices and microprocessors. The waveforms and captured data supported the design implementation and predicted timing waveforms. The benefit of work on this design is the utilization of higher speed COTS microprocessors in space applications and a testbed for investigating software fault tolerant methods.

ACKNOWLEDGMENTS

The author would like to take this opportunity to thank all the people who provided the support and assistance that made this work possible.

To Captain David Summers: Your expertise provided great insight to the hardware/software interface requirements in this project. Additionally, your friendship provided the nucleus of a team that I am proud to follow in the footprints of.

To David Rigmaiden and Jim Horning: Your expertise and knowledge was appreciated in overcoming difficulties during the course of this project.

To Professor Alan Ross and Professor Herschel Loomis: Your guidance, patience and tutelage were instrumental in the completion of this portion of the project.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Increasing the radiation tolerance of a spacecraft against the environment of space is the most important aspect in making it more survivable. Since the end of the 1980s, the Defense budget has seen a dramatic decrease in funding. This has in turn affected the research and development of radiation hardened devices by the commercial sector. Additionally, commercial companies such as Intel are reluctant to switch their foundries from production of normal microelectronics devices to radiation-hardened devices as a result of the loss in revenue. The result of this is a very limited availability of hardened devices at a high cost. Spacecraft Engineers working with lower budgets are therefore forced to look for alternative cheaper, faster and better performing methods of increasing the survivability of the Spacecraft. One alternative is the use of commercial-off-the-shelf (COTS) devices in place of radiation-hardened devices. COTS devices present spacecraft engineers with shorter design-to-launch times, lower parts costs, orders of magnitude better performance, and a wider range of available software than radiation hardened (radhard) devices. The major drawback to utilizing COTS devices in designing a spacecraft is their increased susceptibility to the effects of radiation, both total dose and single event upsets (SEUs) and system design techniques to protect them from this radiation such as increased shielding.

This thesis is a continuation of an ongoing multi-thesis project initiated by LT. Payne [Ref. 1], Capt. David Summers [Ref. 2] and Capt. Kim Whitehouse and LT. Susan Groening [Ref. 3]. It will present the concluding designs and testing of a fault tolerant

1

computer evaluation system including the design of the system controller. Additionally, it will also present the necessary changes for a space flight ready design.

The system is designed to perform two functions. First, it can act as a software testbed by enabling testing of fault tolerant software in the presence of radiation induced SEUs in a test chamber. This allows testing of the software algorithms in the environment they were designed to operate enabling detection and isolation of errors. Additionally, the design can be used as a combination software and hardware fault tolerant computer system. This is accomplished by utilizing the fault masking ability of the hardware with fault tolerant software. Both of these concepts will be discussed further in the body of the thesis.

A. THE SPACE ENVIRONMENT

As satellites become increasingly complex and versatile, the amount of electronic equipment in them grows. Care has to be taken to protect this equipment from both temporary and permanent damage from the environment. Designing equipment for space requires that the designers know the working environment. Like any environment, space dictates the characteristics of devices intended to operate there and imposes requirements on any equipment that would function there. This environment poses a risk to all earth orbiting satellites and missions to other planets in the form of electromagnetic radiation from the sun: not only visible light, but the entire range from radio to gamma rays. In addition, it is also filled with the corpuscular radiation of the sun, the solar wind. Some of this is trapped within the earth's magnetic field forming the intense radiation of the Van Allen Belts. [Ref. 1]

2

1. Radiation

Radiation is the movement of energy through space by propagation of waves or particles. Most of the radiation in space near Earth comes from the sun, as fusion in the sun shoots particles through space. Around the planet's magnetic field, these particles become trapped or are deflected away from the planet. These particles pose a threat to the equipment of a spacecraft and can cause damage or disruptions in microelectronic devices. [Ref. 1]

These particles are either ions or photons. When an atom is hit by a fast-moving particle, an electron can be torn off producing an ion. There are two types of ions: light and heavy. The proton or light ion is the simplest positive ion and is a fundamental particle with low mass. The heavy ion or alpha particle is produced from the Helium atom. The helium atom contains two electrons, two protons, and neutrons. When the electrons are stripped away, the atom is ionized to HE++, which is known as an alpha particle. The classifications of ions as heavy or light is dependent on the atomic number of the element. All ions starting with the element Helium are classified as heavy ion. Unlike ions, photons have neither mass nor charge. X-rays are an example of photon radiation. [Ref. 4]

Multiple sources in space produce these radioactive particles. The first and largest source of radiation is the Sun, which produces solar flares and winds. Solar activity of the sun varies over an 11-year solar cycle, producing a variable average of solar particles. Though the solar activity is predictable on a macro scale, the sun still produces wide variations in radiation intensity on a day-to-day, hour-to-hour basis.

3

A second radiation type is Galactic Cosmic Ray or GCR, which are particles that reach near-earth from outside of the solar system. The cosmic ray consists of heavy ions produced by such events as exploding stars.

The last and largest contributor to a spacecraft's total dose is from particles trapped in the Earth's geomagnetic field, otherwise known as the Van Allen Belts. The belts are a fixed hazard to spacecraft and are distributed nonuniformly within the magnetosphere. Any satellite in orbit is subject to effects from the Van Allen Belts. [Ref. 5]

With this in mind, two factors are calculated to assist in determining the survivability of the spacecraft. The first is the total dosage, the total amount of radiation the spacecraft will be exposed to during its lifetime. The second is the dose rate effect, the amount of radiation the spacecraft is exposed to at a particular time. As the spacecraft orbits, radiation passes through it possibly affecting the spacecraft subsystems. When this radiation interacts with microelectronic devices, it can cause a malfunction known as a Single Event Phenomena, or SEP. Single event phenomena consist of three different effects, the single event upset (SEU), single-event latchup (SEL) and the single-event burnout (SEB), which are discussed in detail in the following section. [Ref. 6]

B.   SINGLE EVENT PHENOMENA (SEP)

SEPs occur when a high-energy particle passes through the microelectronic device and deposits enough charge to cause a transistor to change state. In most cases, the transistor only changes state long enough for the charge to be absorbed back into the system and then resumes its original state. The transistor's state change can lead to

4

latchup in parasitic transistors, high current state in a power transistor, or can be latched into a storage element. These three main types of SEP in Complimentary Metal Oxide Semiconductors (CMOS) are discussed in the following sections. [Ref. 6]

1. Single Event Upset (SEU)

An SEU is an unpredicted change of state or "bit-flip" induced by an energetic particle such as a proton passing through a device. In a spacecraft computer, for example, a bit-flip could lead to a random change in critical data confusing the processor to the point it crashes. In microprocessors, SEUs are typically grouped into one of two error types: program run errors and data errors. Program run errors are errors that occur in the control logic, program counter (PC), or any other register that determines the state of the processor. Data errors are typically confined to the data registers and cache. These two types of errors are not necessarily exclusive. A data error could occur in a register that is later used as program address. When the microprocessor reads the address held in that register it is in the wrong location and begins to execute incorrect code. [Ref. 6]

2. Single Event Latchup (SEL)

Integrated circuits are made by combining adjacent p-type and n-type regions into transistors. By the nature of the process, parasitic transistors are formed along alternate paths through the circuit. These parasitic transistors are biased off by the circuit design under normal circumstances. Latchup occurs when a charge, such as that produced by a particle, activates one of these parasitic transistors, which forms into a circuit with large positive feedback. This creates a short circuit across the device, with two possible outcomes. The first is the current drawn through the parasitic transistors generates more

heat than the device can dissipate and destroys it. If the device is able to dissipate the heat, the large amount of current drawn through the parasitic transistors prevents the circuit from working correctly, which is a non-destructive SEL. The normal symptom of a non-destructive SEL is of a hung system, which requires the system power to be cycled before proper operation is restored. [Ref. 6]

3. Single Event Burnout (SEB)

Single Event Burnout is another condition that can cause device destruction. It is caused by a single ion, for example from a GCR, which induces a high current state in a MOSFET destroying the circuit. [Ref. 6]

C. COMMERCIAL-OFF-THE-SHELF VS. RADIATION HARDENED DEVICES

The radiation effects discussed in the previous sections, with the exception of SEUs, are destructive in nature. The main way of reducing their effects is by utilizing radiation hardened (radhard) devices or providing shielding. A radhard device is one that is specifically designed to be able to withstand higher amounts of radiation than standard commercial parts.

Determining the suitability of commercial-off-the-shelf (COTS) microprocessor for space applications is a subject of ongoing research. There are multiple reasons for utilizing a COTS product within such a harsh environment as space. This section will present a few of the rationale leading to the use of COTS.

1. Forward-looking Technology

As touched upon earlier in the introduction, the United States radiation hardened market is rapidly shrinking. The small percentage of the overall market that requires radhard components puts severe economic constraints on the companies that produce these devices. The number of companies developing and marketing radhard devices is rapidly on the decline and the remaining companies are not developing new chip designs. For these reasons, the development of radhard devices is lagging behind state of the art technology by two or more generations. As an example, a spacecraft launched in space at present would have at best the equivalent of a 486 66 MHz CPU radiation hardened microprocessor compared to the standard home computer with a modern 700 MHz Pentium III processor. This entire order of magnitude difference in processor capability makes the COTS processor especially appealing. [Ref. 7]

2. Faster Design-to-Orbit Time

Parts availability is crucial in maintaining a development schedule. The limited availability of radhard devices offered by many manufacturers can lead to a delay in production schedule. By utilizing COTS devices, the production flow is maintained. The spacecraft engineer is given a wider selection of devices to utilize from multiple vendors. Additionally, the utilization of COTS allows for parts interchangeability in case of failure. This translates to less non-value-added time in the development schedule. Numerous companies are conducting radiation testing on devices, creating a growing database of devices suitable for space applications. [Refs. 7 and 8]

7

3. Reduced Cost

Low demand and little profit exist in the production of radhard devices, which has led to many manufactures abandoning their production of radhard devices in favor of the more lucrative, higher volume consumer electronics. The limited availability of these devices then leads to an inflation of the cost. Part cost directly impacts the cost of the product. In a time of shrinking budgets, the spacecraft engineer is looking for a cheaper suitable product. The best alternative is the development of hardware and software fault tolerant designs with non-radhard COTS. [Ref. 7]

D. PURPOSE

The goal of this research is the testing and implementation of a fault tolerant computer system using COTS microprocessors that is capable of operating in the presence of radiation induced SEUs. This thesis specifically concentrates on the programming and initial testing of a design previously fabricated in the work reported in Reference 2.

This design did not take into account total dose radiation, which is a factor that usually limits the operational lifetime of spacecraft electronics. This factor is determined by the electrical properties of solid-state components exposed to radiation over a period of time. Ultimately the long exposure to radiation leads to changes in the component parameters outside of design specifications and causes the circuit to cease proper functioning. This factor is less stringent in the design because of spacecraft shielding, component selection and survivability.

Successful completion of this project will lead to numerous benefits for the space community. First, the adage of faster better cheaper can be utilized in the development of spacecraft. The spacecraft engineer will have a broader choice of devices and software to choose from at a reduced cost. The spacecraft design will no longer be restricted to the use of radhard components.

Second, the fault tolerant system can be utilized as a testbed to analyze software fault tolerant programs. The fault tolerance hardware is able to detect the SEU and log the time and kind of an upset. The software can then be observed in the manner in which it handles the error. This testbed will allow the testing of software in a simulated space environment prior to use in orbit.

Last, the system can be utilized as a hybrid fault tolerant computer system. In this configuration, the processor is additionally monitored for SEU. Upon detecting an upset, the processor is restored to the state prior to the upset. The processor then continues execution from the point prior to the upset with little downtime and no loss of data. This is dramatically different from current operations where a processor is reset when an error occurs, resulting in downtime, loss of data and spacecraft availability. As shown, this is a major advance in the handling of spacecraft system failures.

E.   THESIS ORGANIZATION

The organization of this thesis follows the design approach used in developing the system. Chapter I has been a brief introduction of the environment in which the system will be operating. Chapter II is background material on research that has led to the foundation and fabrication of this design. Chapter III contains the programming, testing,

and implementation of the programmable elements of the system. Chapter IV presents the design and programming of the system controller. Chapter V presents the final steps in design completion. Chapter VI presents the steps that were taken in testing the design after manufacture. Chapter VII presents steps required to transition the current test bed design to a flight ready design. Chapter VIII presents the conclusions developed during this research and discusses topics for follow-on work.

II. BACKGROUND

Fault tolerance has been implemented in computers for many years. A digital system, though very reliable, does not operate fault free. When a system experiences a fault, it has to be detected and corrected. The technology of computer systems has progressed at a rapid rate and many fault tolerance requirements have been dropped in order to improve speed or performance. However, the Department of Defense requires the use of fault tolerant designs in systems that perform critical tasks, such as the control system of the F-117 stealth aircraft. A minor fault in the computer during flight would mean disaster for the aircraft. This level of performance has maintained the practice of fault tolerance methods at the forefront many of designs. [Ref. 8]

The purpose of this chapter is to provide the reader with a brief background of this project. The chapter starts by outlining the general concept of fault tolerance and focuses in on the design and implementation of this system.

A. FAULT TOLERANCE

There are two approaches to increase the survivability or reliability of electronics in a spacecraft, which are radiation hardening and the use fault tolerance. Figure 2.1 provides a flow diagram for the design of a reliable system. The first method, radiation hardening of devices, is simply constructing devices in such a way as to increase the total dose survivability and reduce the possibility of an SEP. Four basic ways to harden a device are with junction isolation, dielectric isolation, silicon-on-sapphire devices, and silicon-on-insulator devices. This method would relate to the left-hand side path

designing a system with fault avoidance by utilizing parts with a high reliability. This system design has increased radiation tolerance, but offers little or no redundancy.

The second method, fault tolerance, follows the right side of the figure and is simply the ability of the spacecraft to functionally operate in the presence of a fault. Fault tolerance is usually achieved by increasing the redundancy of onboard systems. Reliability is determined by the design of the system, the parts utilized, and the operating environment. One method of increasing reliability is by employing the worst-case design, using high quality components, which in turn adds cost. An alternative method of improving spacecraft reliability is to use a fault–tolerant design. Fault tolerance can be accomplished in either software or hardware. This section will discuss the redundancy methods that are relevant to this design, which are time, software, passive, and hardware redundancy. [Refs. 9 and 10]
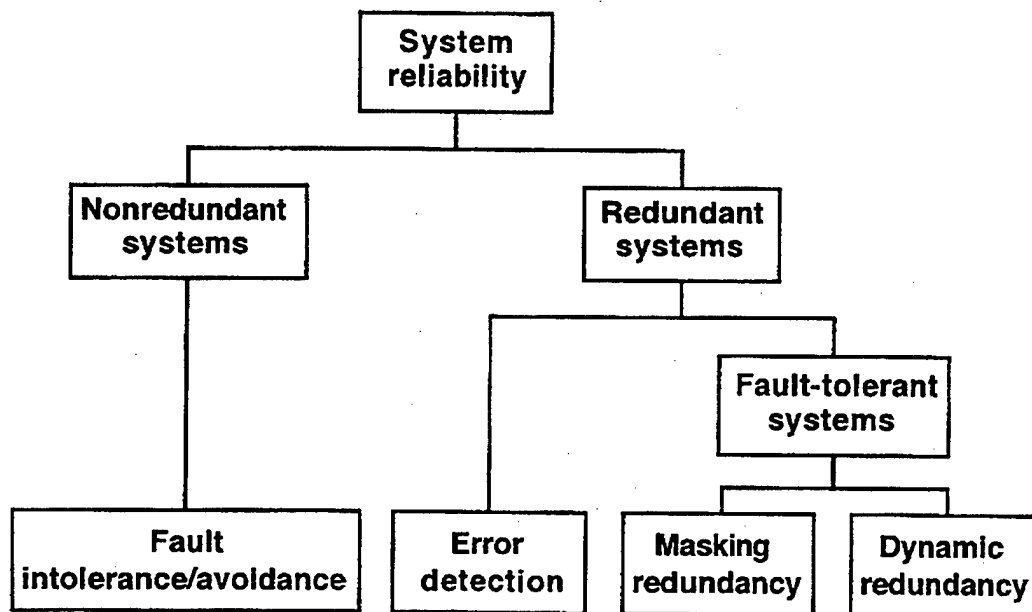
Figure 2.1. Strategies in Designing a Reliable System. From Ref. [9]

1. Time Redundancy

Time redundancy is one of the easiest methods to implement; it involves the restoration of a system to the point immediately after experiencing a fault. This fault is detected by placing checkpoints and with a timeout mechanism. If the system fails to perform a task within a certain amount of time, a fault is detected. The restoration of the system is accomplished by rollback of instructions, segments of programs or entire programs to the last checkpoint. The problem with this method is that it can be time consuming, which is determined by the size of the program and memory that is restored. Additionally, there is a loss of information to the point that the system last saved data. [Ref. 10]

An alternative method of time redundancy is the performance of a calculation numerous times for accuracy. This requires the system to save the state before the calculation, perform the calculation and save it, make a context switch back to the beginning of the calculation, perform it again, and then compare the results of the two different calculations. This results in a large computational drain on the system and two-fold increase in calculation time. [Ref. 10]

2. Software Redundancy

No matter how capable the programmer, almost all software contains faults. A way to achieve some level of protection from these faults is the implementation of a software redundancy method. One such method is N-version programming, which is the addition of software modules to provide checks. For example, five individual programs

are designed for the same function. They are all executed, and their outputs are voted upon. Additional methods of software redundancy are consistency checks of the data against known correct values and capability checks to ensure those functional programs are operating correctly. [Ref. 10]

A subset of software redundancy is error-correcting codes. These codes can be utilized to provide automatic fault detection. One of the best-known codes, the Hamming single error correcting code, is used to increase reliability of information transmitted or stored in memories. [Ref. 10]

3. Passive Redundancy

Passive redundancy employs multiple units, some of which are not continuously operating and are command selectable. In this configuration, redundant items act in response to a specific failure or anomaly. The detection of a fault is achieved by conducting periodic tests, self-checking circuits, or watchdog timers. Passive redundancy allows mission operations to continue in the presence of one or more failures. [Ref. 10]

4. Hardware Redundancy

The most widely accepted view of hardware redundancy is the addition of components. Hardware redundancy can be broken into two subcategories: static and dynamic. Static redundancy, also known as masking redundancy, is the addition of extra component to mask out a fault near instantaneously. One of the major methods utilized to accomplish this is Triple Modular Redundancy or TMR. Dynamic Redundancy is implemented by monitoring the operation of the numerous devices for a fault. In this

system, only one module or device is operating at a time. If a fault is detected in this operational device, it is switched and replaced by another device. [Ref. 10]

The design described in the following chapters of the paper and employed in this system is Triple Modular Redundancy, a hardware redundancy technique. The TMR concept is implemented by utilizing three identical modules that feed their output to a voting unit. This voting unit then compares the outputs and passes the majority vote to the output, essentially masking out any single fault.

a) Triple Modular Redundancy (TMR)

As stated before, TMR is implemented by the replication of the devices and performing a majority vote to determine the output of the system. For example, if Module A becomes faulty, the two remaining module's outputs mask the fault when the majority vote is performed. The inputs and outputs of a module do not have to be single bytes. A word can be inputted into a module to produce a word output. This word has to then be inputted into parallel voting units to vote. The basic concept of the TMR circuit is shown below in Figure 2.1

The concept of TMR can be expanded to include multiple voting modules to produce an N-modular redundant system. As N gets larger, the logic required to realize the circuit and the added levels of delay get excessive. The typical range for N is from three to five. [Ref. 10]
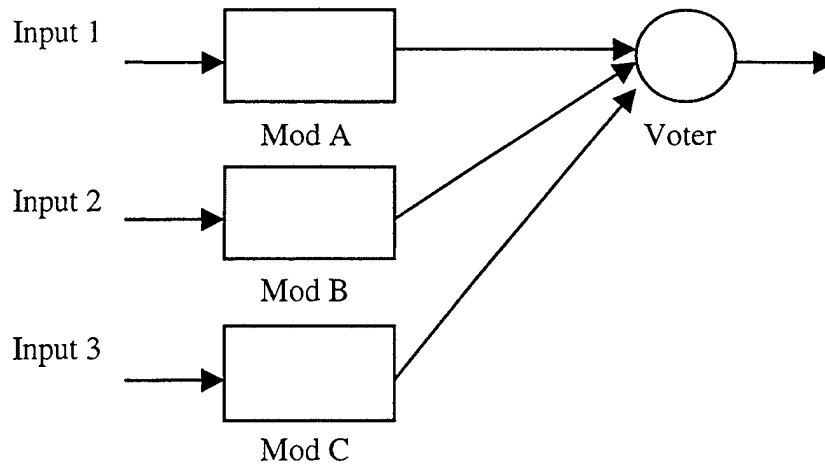
Figure 2.2. Basic TMR Circuit Implementation. From Ref. [1]

The TMR system does have drawbacks, the primary being that the voter is a single point of failure. If the voter fails for some reason, the system will crash or propagate errors. A method to prevent this problem is the use of triplicated voters. [Ref. 10]

B.   TMR MICROPROCESSOR DESIGN

The framework for the system in this design was first developed and simulated using Verilog by Lieutenant John C. Payne, Jr., USN, as a Fault Tolerant Computing Testbed [Ref. 1]. Following this Captain David Summers, USMC implemented and fabricated the design [Ref. 2]. The remainder of this section is a brief synopsis of the TMR Testbed Design.

16

1. Hardware Design and Operation

The first step in the design process was the integration of the system components. As stated previously, TMR is implemented by on the replication of three modules. This design was first focused on the 3081 as a single system and then triplicated. Figure 2.3 demonstrates the implementation of this concept. The TMR implementation has relatively few changes from the single processor design. The major additions to the design were the data, address, and control voter components.

As shown in Figure 2.3, the three system processors are connected in parallel. The system acts as if only one processor is present in the system. The processors perform functions in a lock step manner from initial boot up by executing the same instructions. The processors then route address, data, and control information through busses to their respective voters. The voters perform a majority vote on the signals and pass them on to the Memory Space and Memory Controller as in a single processor system. If an error is detected in a voter, the Memory Controller generates an interrupt.
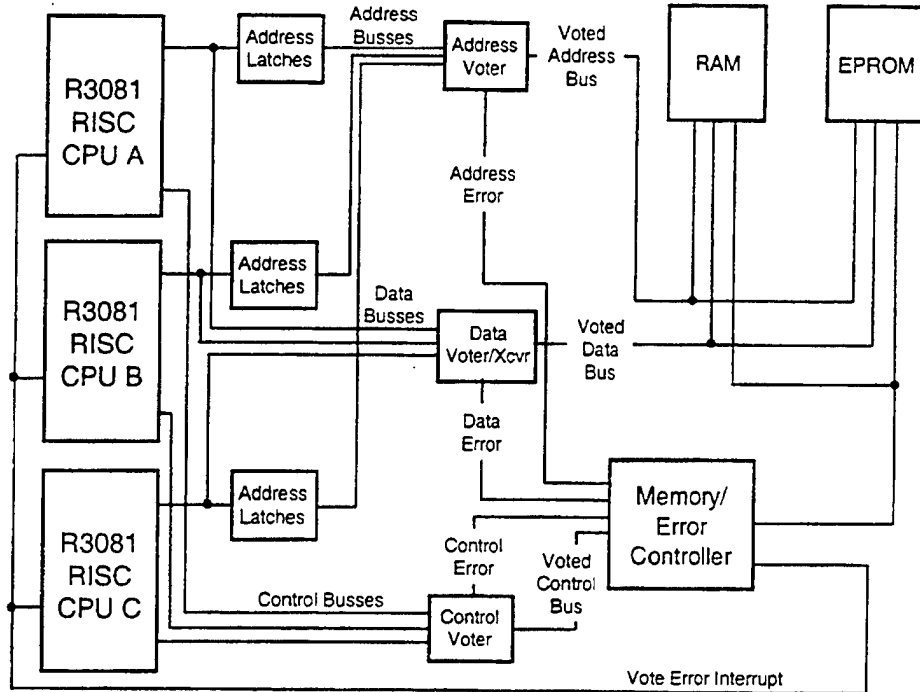
Figure 2.3. TMR R3081 Board Design. From Ref. [1]

2. Fault Detection

Though the voters mask out the fault generated in the data going to memory, the problem remains of detecting which processor was at fault and where. In order to accomplish this, the internal registers of each processor have to be stored and examined. The information (address, data, and control) is captured prior to being voted by placing First-In-First-Out (FIFO) Registers on the address, control, and data busses between the processors and voters. If an error is detected, by any of the voters, the current bus cycle completes and an interrupt is generated. The processors are then restored to the state prior to the fault and resume operation. The arrangement is shown below in Figure 2.4.

18

This design protects only the processor operation and the processor output. The reliability of the data stored in the memory is not improved. This issue is discussed in Chapter VI as a part of the required preparations for a space flight design.
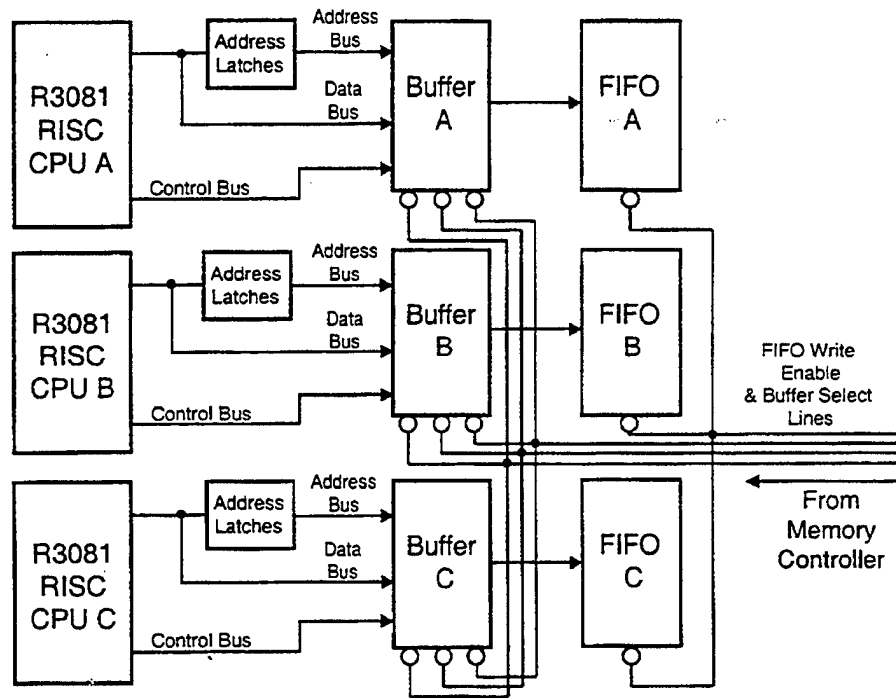
Figure 2.4. TMR FIFO Interface. From Ref. [2]

C.   DESIGN IMPLEMENTATION

Lt. Payne's design and simulations were the framework for the concept of the TMR system. His design product was software verification in Verilog of this implementation of TMR. The thesis presented by Capt. David Summers [Ref. 2] describes the implementation of the TMR design in hardware and the required changes. The following sections will provide a brief overview of these design changes and further information regarding them can be found in Reference 9.

1. Design Hardware Changes

The process in the design of any system is driven by many factors including part availability and compatibly. Capt. Summers was required to implement design changes in order to provide a working board for future test and space applications. The three major changes implemented were the addition of a system controller FPGA and I/O interface ports. The system controller FPGA was added to replace some of the functionality provided by the computer in the Verilog design. The I/O interface was added to provide a method to upload programs and control the board during testing. The design and implementation of the system controller FPGA is covered in this thesis. Additional support elements such as oscillators, buffers, and a reset circuit were also added to the design. These design changes are highlighted and shown in Figure 2.5.

2. Final Design

The design and manufacture of the TMR board were completed by Capt. Summers, but he was unable to complete the programming, test, and verification of the design. This thesis continues the preparation of the board for eventual cyclotron testing and spacebased applications. This author began work on the TMR design with Capt. Summers on the design of the PLDs, FPGAs programs and the detailed timing analysis. This joint effort was presented in Capt. Summer's thesis as Chapter IV. Chapter III presents the continuation of this initial effort with the programming and testing of the Programmable Logic Devices (PLD).
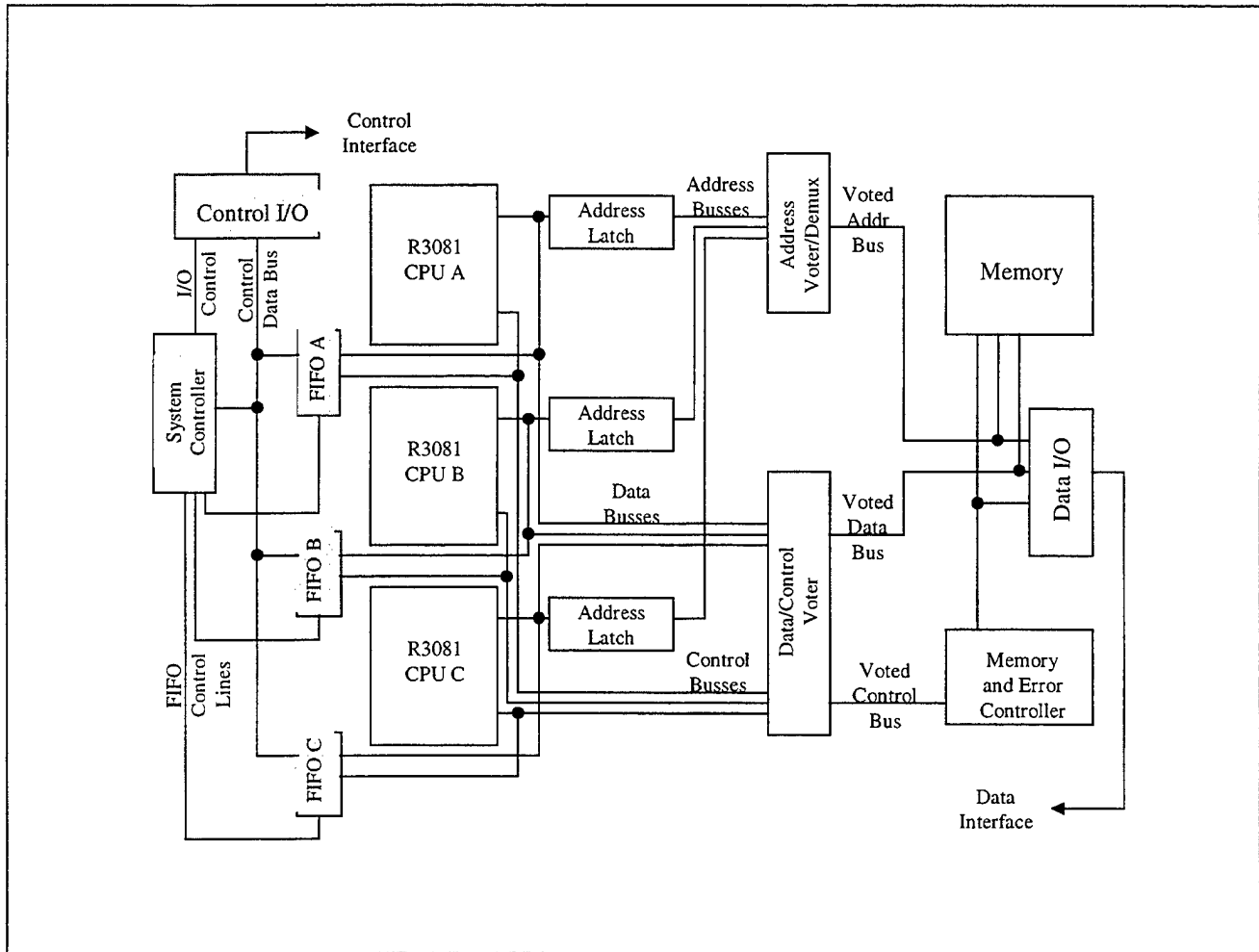
Figure 2.5. TMR R3081 Block Diagram. After Ref. [2]

THIS PAGE INTENTIONALLY LEFT BLANK

III. PROGRAMMABLE LOGIC DESIGN AND TESTING

Designers are continuously challenged to add more logic to system while using the same amount of board space. In order to incorporate this in the TMR design, the Memory Controller and Memory Enable functions were implemented in Programmable Logic Devices (PLDs), and Field Programmable gates arrays (FPGAs) were used to implement the data, control, and address voting logic. The programmable logic components of the design are identified as shaded blocks in Figure 3.1.
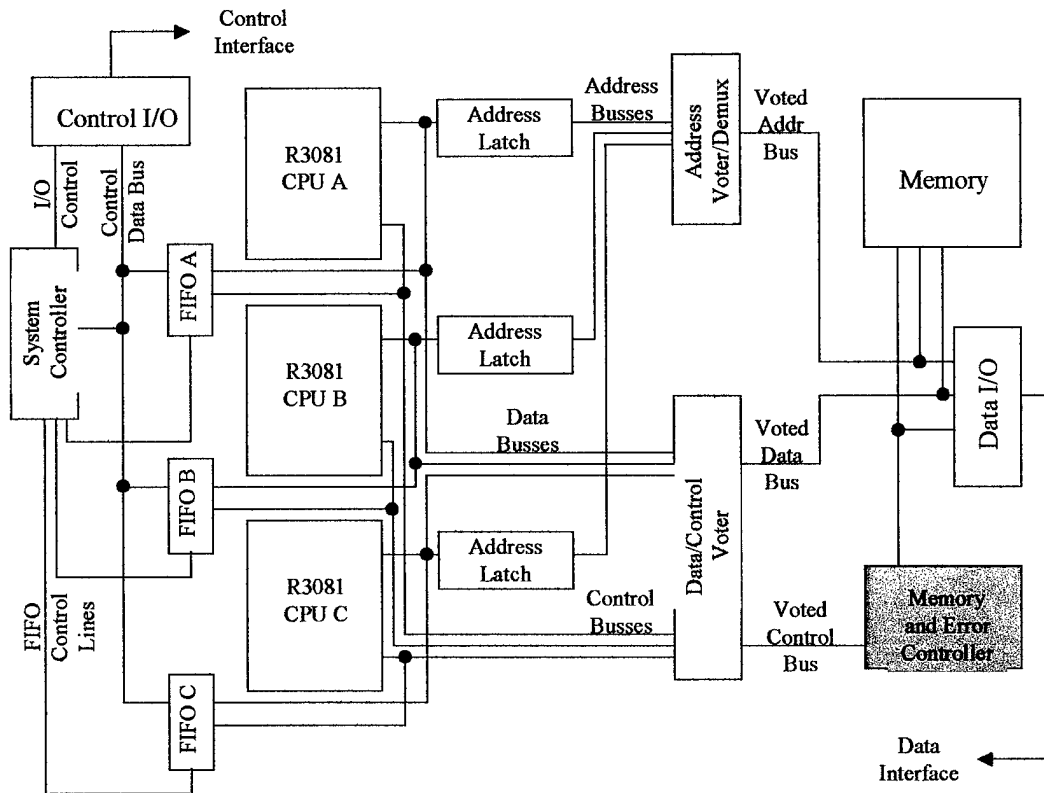
Figure 3.1. Programmable Logic Device Identifier. After Ref. [2]

A.   PLD PROGRAMMING

The final system design shown in Figure 3.1 utilized four user-defined modules, which are the address/voter demux, data/control voter, system controller and the memory and error controller. The first three modules were implemented utilizing Xilinx FPGAs are discussed in later sections. The memory and error controller modules were implemented in the TMR design utilizing two Atmel ATF22V10C-7PC PLDs. Detailed information regarding device selection and program logic is found in Reference 2 Chapter IV. This chapter will describe the steps taken in the programming and testing of these devices.

The software program utilized to construct files for download to the PLD is a program called WinCUPL, which is a Windows version of Universal Compiler for Programmable Logic (CUPL). CUPL is a HDL (Hardware Description Language) comparable to ABEL used to program logic devices. The input to WinCUPL is a user created Programmable Logic Device (PLD) file and the output is a standard JEDEC file used to program the device. After compiling, the JEDEC file is downloaded to a programming unit to program the device fuses. The program device utilized in the design of this system is the ALLPRO 96 256 Pin programming system by Logical Devices, Inc. After the file is downloaded, it is programmed into the device and the logic output of device is compared against test vectors. The following sections give a brief synopsis of the purpose of the devices and the testing and verification procedures.

1. Memory Controller PLD

The main function of the memory controller is to provide all of the signals necessary for memory access during bus cycles. The memory controller inputs consist of the chip select signal for all peripheral devices such as RAM, ROM, timer, UART, and voter interrupt. Additionally, the voted signal such as read, write, data, address, and control vote errors are inputted to the memory controller. These inputs are utilized to generate the vote interrupt, read/write cycle enable, acknowledge, bus error, and cycle end signals. Appendix A gives a detailed pin out of the Memory Controller PLD.

2. Memory Enable PLD

The Memory Enable PLD assists the Memory Control PLD during bus transactions. It accomplishes this by producing the read and write enable strobes for the memory system, read and write data enable signals to control the drivers on chips between the busses, and a positive and negative logic synchronous reset. Inputs to the PLD consist of the voted read/write and voted byte enable signal. Utilizing these inputs, the PLD outputs the write and read enable signals for the peripheral devices.

B. PLD TESTING

Test vectors allow the designer to verify, test and debug a PLD design for proper functionality before it is used in the system. Though the test vectors can pass when tested on the CUPL functional simulation, the same vectors may fail when tested on the PLD programmer. This is the case with the memory controller PLD that had two vectors fail. Two reasons that this could occur are improper test vector usage and a result of the programmer's hardware characteristics. The programming hardware dictates the

sequence in which inputs in a given vector are applied to the device. Analysis of device output and logic equations is the only manner by which to analyze and correct this. Analysis of the device output led to the conclusion that the PLD was unable to latch the counter causing the vector failures. The PLD program file was modified to correct this error and passed both the simulation and programmer tests. The impact of the change to the PLD functionality is that the bus error and cycle end signal will assert on one clock earlier. This is no manner affects the function of the TMR system. After completing the program changes, the file was compiled and downloaded into the programmer. Programming of the PLD passed all test vectors successfully. An updated design file is given in Appendix A.

C.   FPGA PROGRAMMING

The field programmable device chosen for this system is the Xilinx XL4013XLA. These devices are utilized to implement address voter\demux, data\control voter, and the system controller shown in Figure 3.1. Reference 2 Chapter IV gives a detailed discussion on the selection of this device and the design of the programs. The software used to develop the program for the devices was Xilinx Foundation Software. The FPGA design is a 3-step process that consists of the following stages. First, design entry, in this stage of the design flow the design is created either in schematic editor or hardware descriptor language (HDL). Second, design implementation by mapping the file to a specific Xilinx architecture, and placing and routing the design, the design file created in the first step is mapped into a physical file format. The physical information contained in this file is then used to create a bit stream file for programming in a programmable

device. The final stage of the process is completed when the bit stream file is formatted by a PROM formatter into a configuration file for the FPGA device that can be stored in a PROM. This is accomplished by converting the BIT file into one of four PROM formats: MCS-86 (Intel), EXORMAX (Motorola), TEKHEX (Tektronix) or straight HEX format.

In a joint effort with Capt. Summers, the first two steps of this process were executed. [Ref. 2] While executing the third step, it was discovered that the PROM code formatter in this software is targeted for Xilinx parts and the Serial EEPROM utilized in this design was an ATMEL AT17C512. Further research into the impact of utilizing this file in the device was conducted by contacting Xilinx and Atmel. After numerous conversations with the two companies, it was concluded that the Xilinx software program uses the device selection in determining the allowable size of the prom file only. Therefore, the difference in device was acceptable and would have no impact. Completing the process the file was formatted into MCS-86 and TEKHEX, then downloaded to the ALLPRO 96 programmer for programming into the Serial EEPROM.

In order to accomplish programming of a device, the ALLRPO has to first recognize the device type. The device programmer was unable to recognize the ATMEL devices for programming and thus unable to program the serial PROM. Research into possible causes for this failure led to a notice on the ATMEL website [Ref. 11]. This notice stated that some ATMEL serial EEPROMs failed to accept device encoding during manufacture and listed steps to overcome this failure. The instructions detailed methods of switching off the programmers' device recognition function. After completing the procedure detailed in the notice, the device programmer was still unable to program the

27

device. An ATMEL FPGA engineer, who assisted with the information for the programmer was aware of the difficulties experienced in programming the devices and offered assistance with programming the devices. The devices were returned after programming for installation in the system. Verification of successful programming is only indicated by the programmer's comparison of the contents of the programmer memory to the device contents.

The programming of the Memory Controller PLDs and the Voter FPGAs allow the system to function as a stand-alone computer system able to detect and correct single bit errors occurring in any of the processors. The design of the system controller FPGA though mentioned in the start of this chapter was not discussed. The design and programming of this programmable device completes the TMR design and is discussed in detail in following chapter.

IV. DESIGN OF THE SYSTEM CONTROLLER FPGA

As stated earlier, programmable logic offers flexibility to the system designer and is essential to performing controlling functions in the TMR 3081 design. The Data Voter and Control voter were previously implemented and programmed using the schematic editor in the Xilinx software. The System controller FPGA presents a more difficult design problem because the logic implemented by this controller cannot easily be modeled as a combinatorial design. Therefore, it was decided to use Xilinx Foundation HDL editor and State Machine tool to design the controller. The system controller performs numerous functions in the TMR system. It initializes the Control UART, enables the interrupts for the 3081 microprocessor after resets, transfers and collects FIFO data, and defines the mode in which the TMR system functions. In examining all of these tasks, there are three distinct possibilities of data flow. These are:

- The FPGA outputs data to the UART
- The FPGA enables the FIFO to output data to the UART
- The FPGA accepts input from the UART

Each of these events is mutually exclusive: that is, no two events can ever occur at the same instant in time. It is the function of the system controller to define, which one of these events listed above is presently in progress. The detailed subsystem programs of the system controller FPGA are listed in Appendix B. The following sections discuss the state machines for each subsystem of the system controller FPGA. Figure 4.1 provides an overview of the flow of the state machines discussed in the following sections.

29
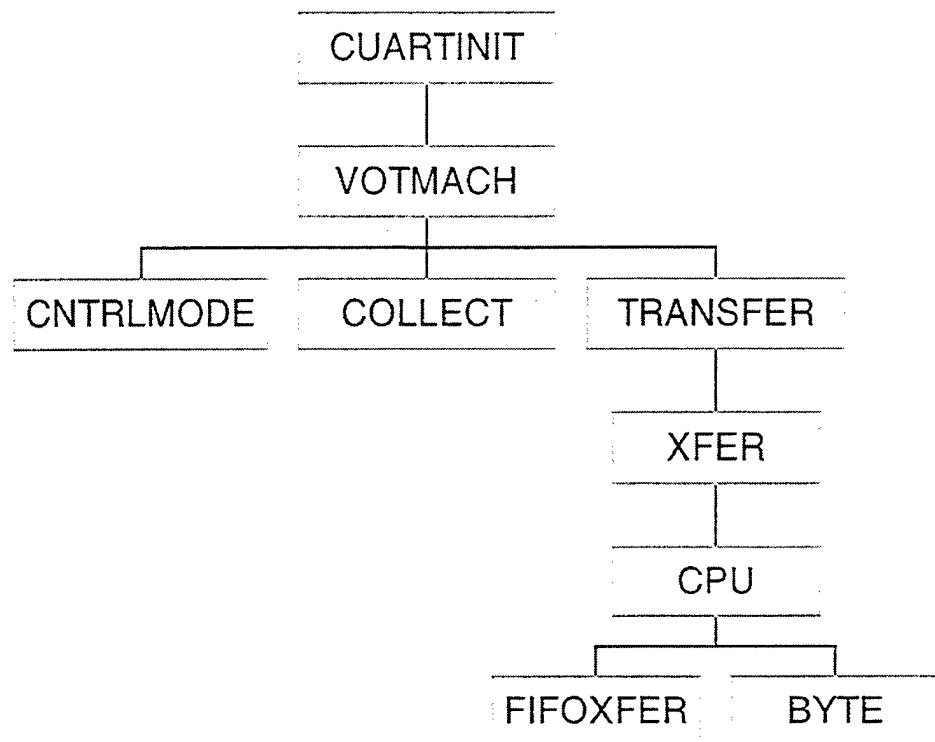
Figure 4.1. State Machine Hierarchy

A. CUART STATE INIT MACHINE

The Control UART Initialization state machine is the uppermost in the state machine hierarchy. It automatically initializes the UART to the proper mode of operation when the system controller FPGA is powered on or reset. The following sections present the operation of the UARTINT state machine.

1. State Machine Functionality

The CUARTIN state machine initializes the control UART to communicate with the HCI. This is accomplished by a sequence of UART register writes. The first register initialized by this sequence is the FIFO control register, which is a write only register.

The register enables the FIFO by asserting the first bit. The next register in the sequence is the Line control register. The line control register controls the format of the data communication. The first two bit in the register are set to '11' setting the serial character word length to 8 bits. The last six bits in the register set the parity and number of stop bits. The Human Control Interface design by Capt. Kim Whitehouse and LT. Susan Groening [Ref. 3] currently has no parameters set for these modes. The default initialization sequence therefore sets the modes for no parity and one stop bit. The Modem control register is next in the sequence and controls an interface with a modem. The assertion of the first two bits and the sixth bit in the register enable the autoflow control mode of the UART. The final registers in the setup sequence are the Divisor Latch (LSB) and Divisor Latch (MSB). These registers are utilized to control the programmable baud generator for the UART. The divisor written into the registers is 72. This divisor sets the baud rate of the UART to 9600. The final state of the UART initialization sequence disable the data lines and deasserts the chip select line. Completion of UART setup enables communication with the HCI and allows the state machine to proceed to the next state machine, VOTE.

The CUARTINT state machine initiates in state S0 and immediately transitions to the next state, S1. The state machine shown in Figure 3.1 contains no conditions for transitioning from state to state. Therefore, once the state machine is initialized it does not depend on any outside input for completion of the control UART initialization process. Continuing in state S1, the FSM asserts UART Enable (UARTEN) and Control UART Chip Select (CUARTCSN). The assertion of these signals enables the UART to

31

drive the bi-directional bus and read status information from a register. In the next state, S2, the FSM first asserts the Control Address bus (CTRLADDR) with the register selection address. The UART has twelve internal registers, only, six of which are utilized during the initialization process discussed in this section. The table below details the register selection dependent on the address placed on the CTRLADDR bus.

DLAB†	A2	A1	A0	REGISTER
0	L	L	L	Receiver buffer (read), transmitter holding register (write)
0	L	L	H	Interrupt enable register
X	L	H	L	Interrupt identification register (read only)
X	L	H	L	FIFO control register (write)
X	L	H	H	Line control register
X	H	L	L	Modem control register
X	H	L	H	Line status register
X	H	H	L	Modem status register
X	H	H	H	Scratch register
1	L	L	L	Divisor latch (LSB)
1	L	L	H	Divisor latch (MSB)

Table 4.1. UART Control Register. From Ref. [12]

In addition to asserting the control address, state S2 places the setup data on the Control Data UART (CTRLDATU) bus that is tied to the bi-directional Control Data (CTRLDATA) bus. This data is written into the selected register and latched in by state S2b with the assertion of the Control UART Write (CUWREN). The next state S3 deasserts CUWREN, which enables the next state, S4, to change the address and data lines without effecting the UART. This process is repeated to setup remaining registers.
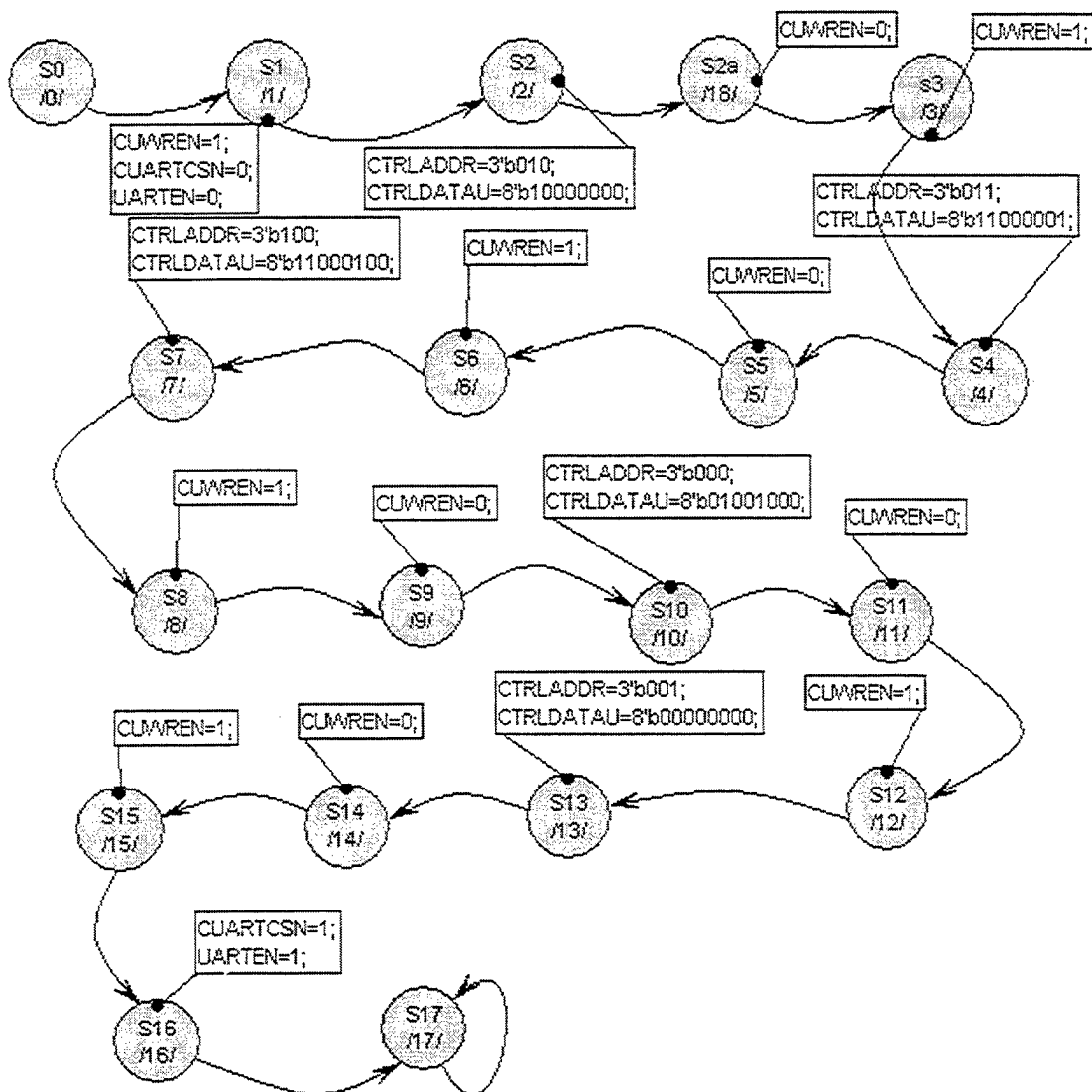
Figure 4.2. UARTC State Machine

B. VOTE MACHINE

The vote state machine is the primary state machine of the system controller FPGA. This state machine has two primary purposes. The first is to detect a voter interrupt from the microprocessor and start the collect/transfer state machines. The second is to detect a Control UART Interrupt (CUARTINT) assertion from the HCI. The

state machine is broken into three functions to ensure control over what state machine is driving the data bus. The following sections discuss the operation of this state machine based on these two functions.

1. Voter Interrupt routine

The main function of the system controller FPGA is to monitor the TMR board for a vote interrupt to collect and transfer the data. This important function is implemented and started with the voter interrupt routine. This state machine detects a voter interrupt and increments an interrupt counter. The incrementing of this interrupt counter is a signal to the collect and transfer state machines to begin the process of collecting the data from the FIFOs and transferring the data to the UART.

The state machine waits in state S1 and monitors the Interrupt Chip Select (INTCSN) signal for indication of a voter interrupt. When, the INTCSN signal is asserted, the state machine transitions to state S2. The state machine then waits for the deassertion of this signal indicating that a voter interrupt has been serviced. The state machine then transitions back to state S1 incrementing the Interrupt Counter (INTRCNT) register, which starts the Transfer state machine discussed in a later section.

The state machine resides in this state until assertion of the CPUDONE signal by the Transfer state machine. Assertion of this signal indicates that the Transfer state machine has completed the transfer of the CPUs FIFO array contents to the HCI. The state machine transitions to state S3 decrements the interrupt counter and then transitions back to state S1 awaiting another interrupt signal. This completes the detection and start of the interrupt service routine.

a) INTRCNTR

The interrupt counter register is implemented in the design to enable the system to contend with numerous consecutive interrupts. The processors will have dumped and restored their internal registers before the system controller FPGA is finished transferring the FIFO data. The state machine controls this by incrementing for each transition of INTCSN signifying to the state machine that an additional interrupt has been serviced. This in turn will ensure seamless operation of the Transfer state machine during multiple interrupts. Additionally, it is important to note that INTRCNTR is a 4-bit register and therefore the maximum number of interrupts counted by this machine is 32. Since the collect and transfer sequences for the FIFO are occurring simultaneously, only an extreme error condition producing consecutive voter interrupts would lead the state machine exceeding this design limit.
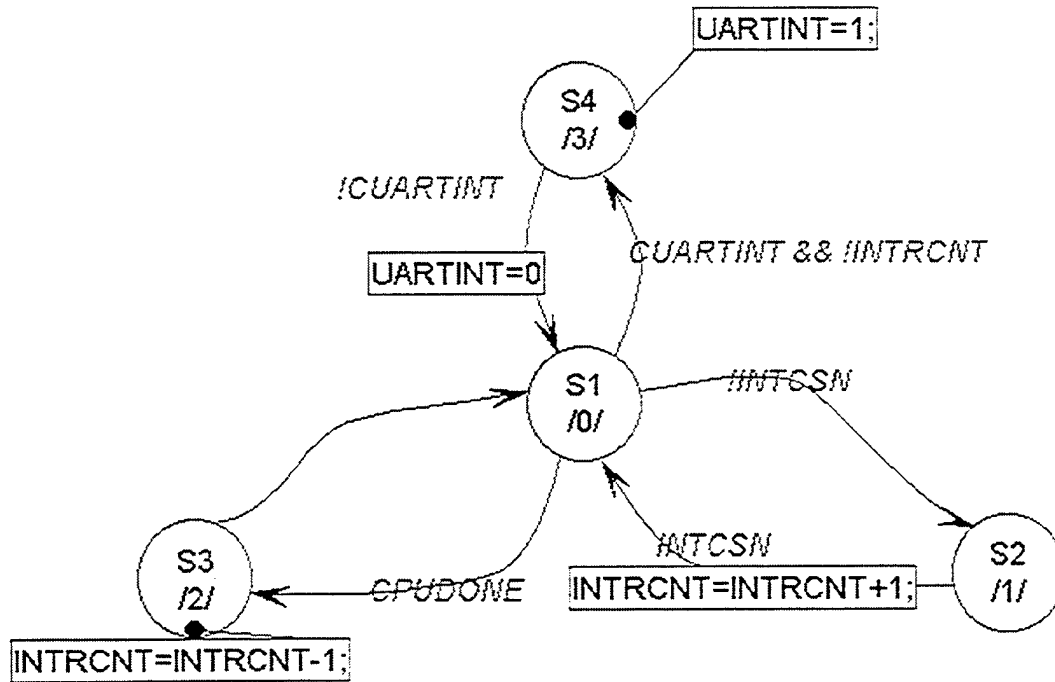
Figure 4.3. VOTE State Machine

2. UART Interrupt Routine

Since the vote state machine is the highest in the hierarchy, it was necessary to implement the detection of a request from the HCI to input data to the system controller FPGA here. The HCI inputs data through the UART to the system controller to implement three functions, which are a board level reset, a system level reset, or to set the mode of operation of the board to one or three processors. The system controller detects this request for input and starts the CONTROL MODE state machine.

An alternate transition path from state S1 to state S4 is caused by the assertion of the Control UART Interrupt (CUARTINT) signal with the INTRCNT register equal to zero. The CUARTINT signal originates from the UART and is asserted by the HCI

36

placing data in the UART. Transitioning to state S4 causes the assertion of the UARTINT signal that starts the CONTROL MODE state machine. This state machine is discussed in the following section.

3. State Machine Design Constraint

An important factor in this state machine design is the imposed constraint that a transition to state S4 from state S1 is inhibited during a voter interrupt service. This was accomplished by the ANDing of the signal CUARTINT and !INTCNTR. This constraint was placed on the design to protect against switching the mode of operation during an interrupt handling routine. An example of this is switching from TMR processor mode to one processor mode during a voter interrupt service. If the processors are currently dumping their registers to the FIFO, the transfer process is asserting and deasserting the CTRLDATA bus. A switch in mode would require that the Read Machine drive the CTRLDATA bus breaking the current FIFO data dump. Therefore, it was decided to block this function from occurring until completion of the service routine.

This basic principle is also implemented in the design of the HCI. During a voter interrupt routine, the HCI is receiving FIFO data and the user is unable to halt the process. The next section discusses the operation of the Control Mode state machine, which the HCI utilizes to setup the TMR system.

C. CONTROL MODE STATE MACHINE

The vote state machine initiates the control mode state machine. This state machine reads in the control word from the UART inputted by the HCI. The state machine next utilizes the control word by checking what bits are set to decide which of

the three functions previously mentioned to perform. This operation of this state machine flow is discussed in the following sections.

1. Mode Initialization

This FSM initializes in state S1 and transitions to state S2 upon assertion of the UARTINT signal by the VOTE state machine discussed previously. State S2 asserts the signals FORCE, mode enable (MODEEN), and CUARTCSN setting the default mode of the TMR design to single processor and enabling the FSM to drive the bi-directional CTRLDATA bus for input. The selection of the default mode of the design to single processor mode is to allow the user to setup the system design and conduct initial tests before transitioning to the TMR mode. This selection can be changed in future designs. The assertion of the final signal CUARTCSN indicates to the UART to activate the asynchronous communications element for data transfer. The FSM next transitions to state S3, asserting the CTRLADDR bus with '000' selecting the holding register (write) of the UART. A transition to state S4 asserts the chip select signal, CUARTADSN, which allows the selection of the control register in the UART. The state machine next transitions to state S5, asserting Control UART Write Enable (CUWREN) and writing the control word onto the data lines. Next, state S6 reads in the data on the input bus AIN tied to the CTRLDATA bus and places the contents into a temporary register MDCTRL. The FSM transitions to state S7 deasserting CUWRENN, CUARTCSN, and MODEEN allowing the other state machines to drive the bi-directional data bus CTRLDATA. State S7 has three possible transition paths that are dependent upon the data in MDCTRL register. The transition paths each contain an 'IF' conditional that checks a bit in the

38

control word which in turn selects the transition path. State S10 causes a change in design

mode from single to TMR processor mode. State S8 and S9 are responsible for a board

and system reset. The Control Mode state machine is shown in figures 4.4 and the

control mode word is given in Appendix B.
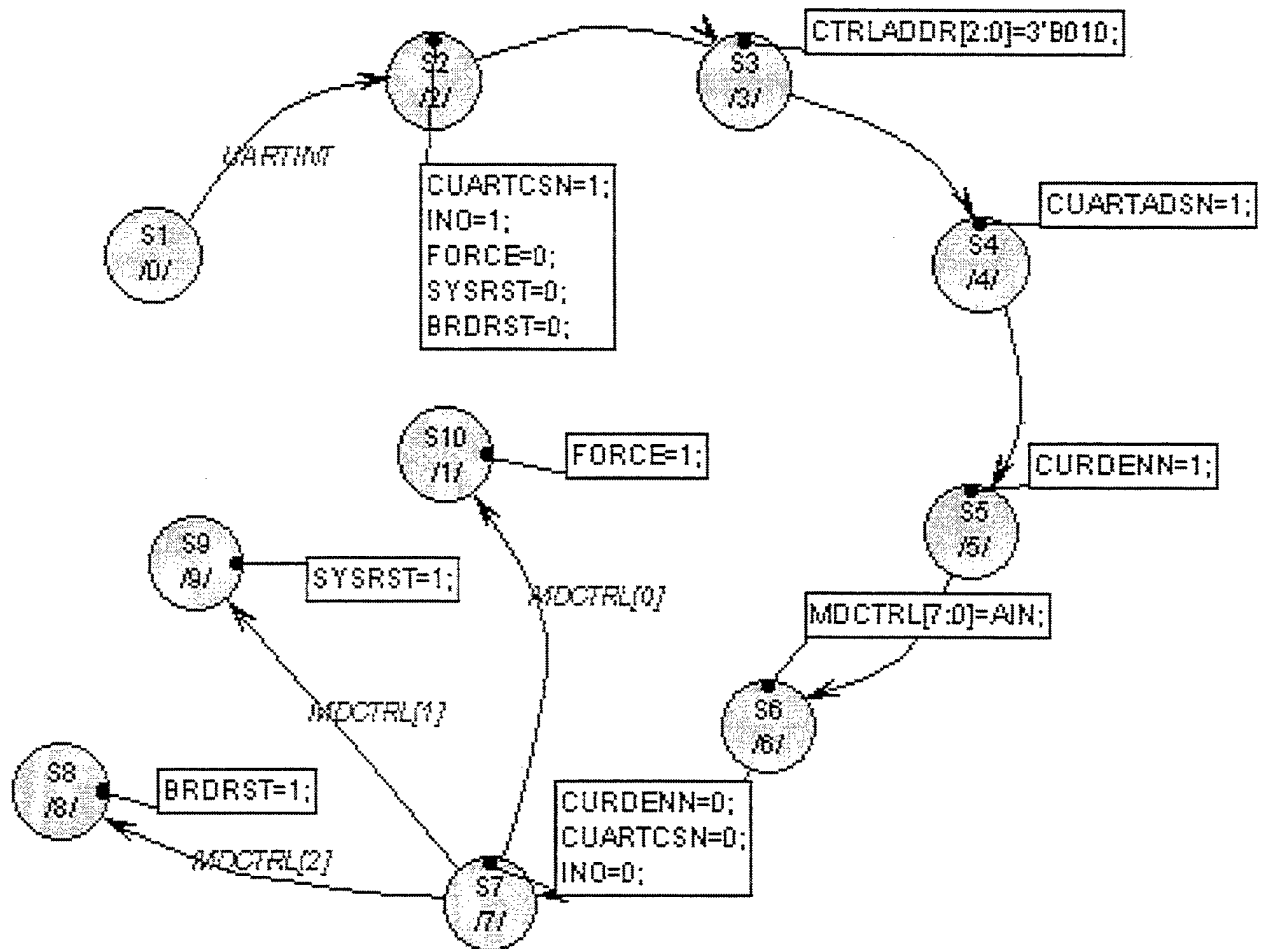
Figure 4.4. MODECNTRL State Machine

D. FIFO DATA COLLECTION STATE MACHINE

The FIFO data collection state machine as the name implies collects the FIFO array data. The assertion of a voter interrupt signal causes the state machine to assert the read enable of the FIFO arrays. A counter increments to 82 ensuring the FIFOs read in the 41 registers data and address saved by the processor. These 32 bits of information are split between four 8-bit FIFOs. The details of this functional state machine are given in the following section.

1. Data Collection

The FIFO data collection state machine resides in state S0 until the assertion of the Voter Interrupt (VOTINT) signal. This signal, asserted by the microprocessor, indicates that a voter interrupt is being serviced and it will remain asserted until the processors have completed the transfer of their registers to memory. Transitioning to state S2, the FSM asserts the first bit on all the FIFOCTRL bus lines. Each FIFO array has a corresponding control line designated as FIFOCTRLA, B, and C. This first line is tied to the Read Enable (RE) pin on the FIFO allowing it to read in data. The FSM next transitions to state S3 which increments the Collect Counter register, COLCNT. This register will increment up to 82 before allowing the machine to transition to state four.

After counting up to 82, the state machine transitions to sate S4 and then to state S1 resetting the collect count. Keep in mind while this is occurring, the transfer state machine discussed in the following section is already transferring FIFO data with the incrementing of the interrupt counter, INTRCNT register by the VOTE FSM.
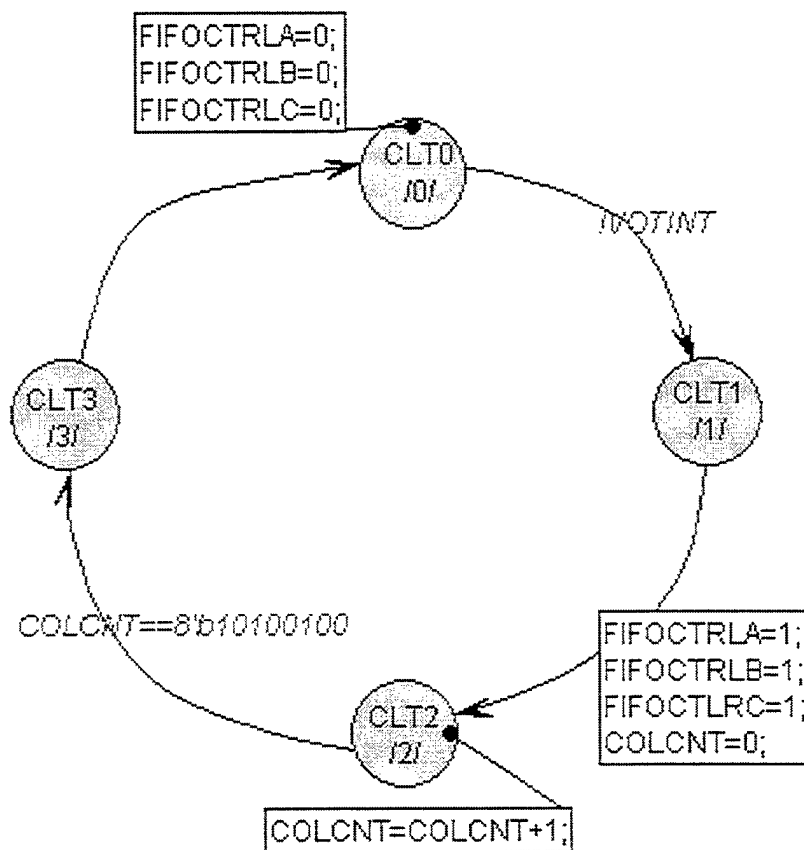
Figure 4.5. COLLECT State Machine

E.   TRANSFER STATE MACHINE

The Transfer state machine is the largest state machine of the system controller

FPGA. This state machine performs two functions first, it transfers a header to the HCI

system to identify which CPU and FIFO is being transferred. Second, it cycles through

the FIFO arrays transferring out the 82 bytes of data and address. In designing this state

machine, it was essential to break the machine into four separate sections. These sections

are the XFER, CPU, FIFOXFER, and BYTE state machines. This section will begin by

the discussion of the XFER state machine, which is the controller of the transfer FSM.

41

1. XFER State Machine

The XFER state machine is the dominant FSM in this design. It monitors the INTRCNT discussed preciously in the VOTE machine section. When the counter is incremented the state machine asserts signal to commence the transfer of FIFO data to the HCI.

With the assertion of INTRCNTR, the FSM transitions to the XFERP state and asserts the Process Transfer (PROCXFER) signal. This signal is used to start the transfer process in the CPU state machine. The final state transition occurs when the INTRCNTR signal equals zeros allowing the machine to transition back to the XFERSTRT state deasserting PROCXFER and switching off the CPU state machine. If the INTRCNT signal is incremented while this state machine is in the XFERSTART state then PROCXFER remains asserted continuing the transfer process for the second interrupt. The XFER FSM is shown in Figure 4.6.
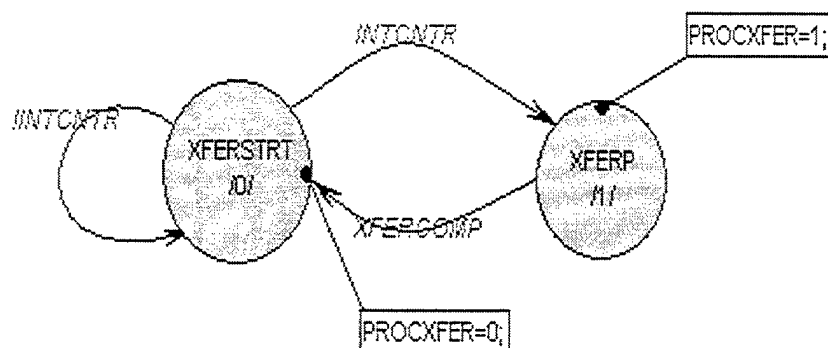
Figure 4.6. XFER State Machine

2. CPU State Machine

The function of this machine is to cycle through each CPU and thereby cycle through the corresponding FIFO array for that CPU. The state machine is started by the XFER state machine and commences with the transfer of CPU A FIFOs. The transfer process begins with the assertion of a signal FIFOENG to enable the FIFO transfer machine and the setting of the CPU header byte. The state machine resides in this state until a counter CPUCNT is incremented indicating completion of the transfer process. The FIFOENG signal is deasserted disabling that set of FIFOs. The state machine then continues the process with the next CPU. The process and state transition is discussed in detail in the following paragraphs.

When enabled by the assertion of PROCXFER, the state machine transitions from the STRT state to the CPU A state asserting the signals FIFO Engine (FIFOENGA) and CHDR[1:0]. The BYTE state machine in forming the Header byte utilizes the Control Header (CHDR) signal. Figure 4.7 shows the header signal format formed corresponding to a CPU.

Bit 1	Bit 0

Processor
00 – A
01 – B
10 – C
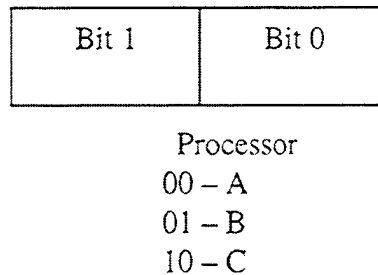
Figure 4.7. CHDR Format

The FIFOENG signal is specific for each processor and is utilized in determination of what transition path to take by the FIFOXFER FSM. The assertion of FIFOENG starts the transfer process in the FIFOXFER and BYTE state machines. The negation of this signal upon exit from a state, for example FIFOENGA transitioning to zero, causes the transfer process for the CPU A to cease.

State to state transition in this machine is dependent upon the signal CPU Counter (CPUCNTR). This signal is controlled by the FIFOEXFER FSM and indicates what CPU is being serviced. When CPUCNTR is equal to 3, the last state in this machine, CPUCOMP, asserts the signal XFERCOMP that indicates to the VOTE FSM that the XFER FSM has completed a transfer cycle. The CPU State machine is shown in Figure

44
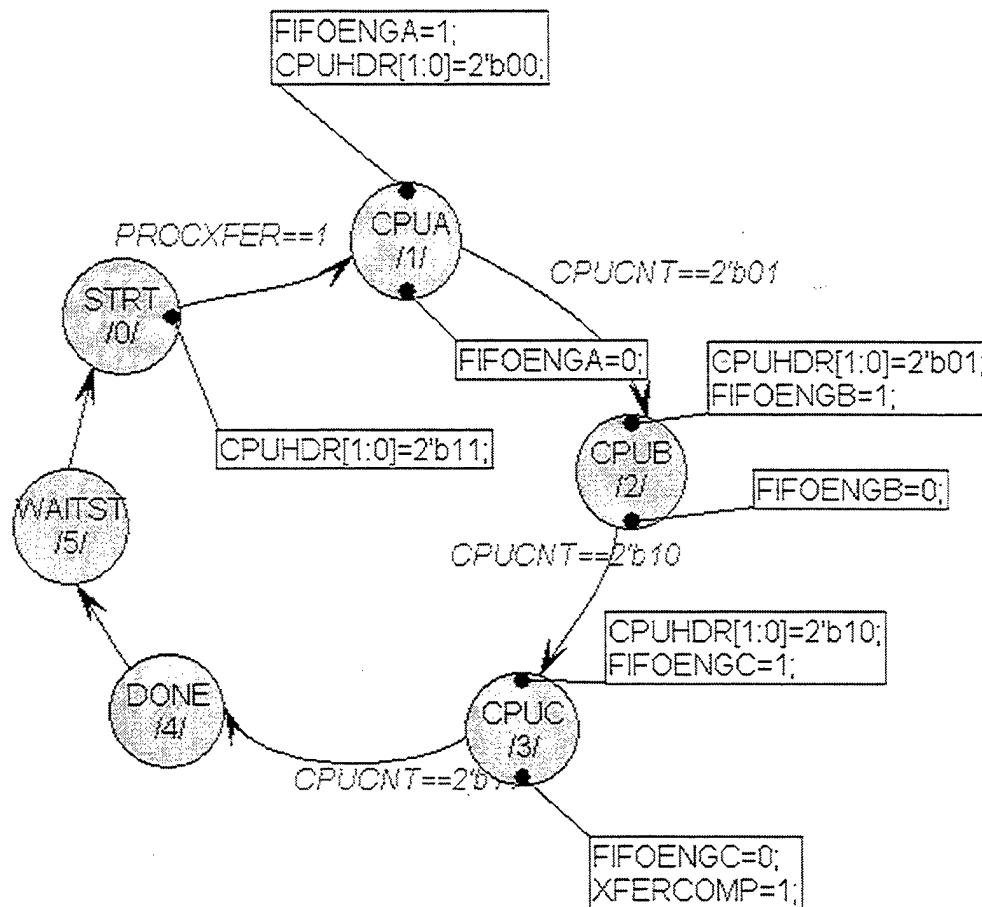
Figure 4.8. CPU State Machine

3. FIFOXFER Engine State Machine

The next FSM in the hierarchy is the FIFOXFER FSM. The purpose of this machine is cycle through the FIFOs array of each CPU. This FSM waits in its initial state FSTART, until enabled by the CPU FSM. When the FIFOENG signal is asserted by the CPU FSM and the Header Complete (HCOMP) signal from the BYTE transfer FSM, discussed next, the FSM transitions to state FIFOST0. The assertion of the HDONE indicates that the header byte has been transferred. This functionality was necessary to

45

allow the FPGA to drive the bi-directional CTRLDATA bus before enabling the FIFO to drive the bus. As discussed previously, the FIFOENG signal is utilized to determine which FIFO control lines are asserted. The transition from state FIFOST0 has three possible transitions determined by the FIFOENG signal, which in turn enable the assertion of three control buses FIFOCTRLA, FIFOCTRLB, or FIFOCTRLC. This design implementation was required to enable the FIFO array corresponding to the appropriate CPU. The FIFO Control bus is 11 bits wide with only the last 10 bits controlled by the FIFO ENGINE state machine. The first bit of the control line is controlled by the FIFODATA state machine discussed previously. Upon assertion of the FIFOCTRLA bus, the BYTE transfer machine begins to transfers the contents of the FIFO that holds the most significant byte of the 32-bit address and data. The format of the FIFOCTRL word is given in Figure 4.9.

Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
FIFO 00 – A		FIFO 00 – B		FIFO 00 – C		FIFO 00 – D		FIFO 00 – E		FIFO WRITE ENABLE
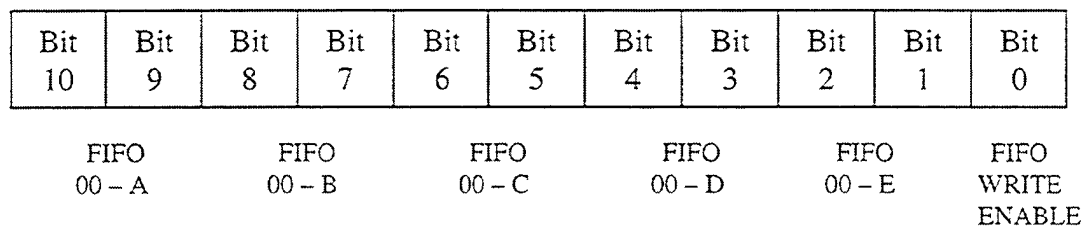
Figure 4.9. FIFOCTRL Word

The FIFOENG state machine waits in state FIFOST0 until assertion of the completion signal from the BYTE Transfer state machine. FIFOST0 additionally asserts the FIFOHDR register that is used in determination of the header byte by the BYTE Transfer machine. The FSM remains in this state until the FIFOCOMP variable is deasserted indicating that the BYTE Transfer machine has completed the transfer of a

46

FIFO's contents. The state machine transitions to the next state when the BYTE TRANSFER machine asserts the signal HDONE indicating that the header information has been sent. The appropriate control lines are asserted during the transition action and this process is repeated until state FIFOST4. This state indicates completion of the transfer of the last FIFO in the array. Upon assertion of the FIFOCOMP signal, the state machine increments the register CPUCNT that is utilized by the CPU state machine discussed previously to enable a different CPU FIFO array. If the CPUCNT is equal to two, the state machine takes a different transaction path and resets CPUCNT to zero. This FSM is shown in Figure 4.10.

FIFOSTART
/0/

FIFOST0
/1/

FIFOST1
/2/

FIFOST2
/3/

FIFOST3
/4/

FIFOST4
/5/

HDRCOMP=0;
CPUCNT=CPUCNT+1;

CPUCNT==2'd1 && FIFOCOMP==1

CPUCNT==2'd1 && FIFOCOMP==1

CPUCNT==2'd0 && FIFOCOMP==1

FIFOEMGA==1 && HDONE

FIFO=TRLA[9:0]=10'b00111111111;

FIFOEMGB==1 && HDONE;
FIFOCTRLB[9:0]=10'b00111111111;

FIFOEMGC==1 && HDONE;
FIFOCTRLC[9:0]=10'b00111111111;

HDRCOMP=1;
FIFOHDR=8'b110000000;

FIFOCOMP==1

FIFOEMGA==1 && HDONE==1 && FIFOCOMP
FIFOCTRLA[9:0]=10'b110011111;

FIFOEMGB==1 && HDONE==1 && FIFOCOMP
FIFOCTRLB[9:0]=10'b110011111;

FIFOEMGC==1 && HDONE==1 && FIFOCOMP
FIFOCTRLC[9:0]=10'b110011111;

FIFOEMGA==1 && HDONE && FIFOCOMP
FIFOCTRLA[9:0]=10'b111111100;

FIFOEMGB==1 && HDONE && FIFOCOMP
FIFOCTRLB[9:0]=10'b111111100;

FIFOEMGC==1 && HDONE && FIFOCOMP
FIFOCTRLC[9:0]=10'b111111100;

HDRCOMP=1;
FIFOHDR=8'b00000100;

FIFOCOMP=1

FIFOEMGA==1 && HDONE && FIFOCOMP
FIFOCTRLA[9:0]=10'b111110011;

FIFOEMGB==1 && HDONE && FIFOCOMP
FIFOCTRLB[9:0]=10'b111110011;

FIFOEMGC==1 && HDONE && FIFOCOMP
FIFOCTRLC[9:0]=10'b111110011;

HDRCOMP=1;
FIFOHDR=8'b00000100;

FIFOCOMP==1

HDRCOMP=1;
FIFOHDR=8'b00000100;

FIFOEMGA==1 && HDONE && FIFOCOMP
FIFOCTRLA[9:0]=10'b111110011;

FIFOEMGB==1 && HDONE && FIFOCOMP
FIFOCTRLB[9:0]=10'b111110011;

FIFOEMGC==1 && HDONE && FIFOCOMP
FIFOCTRLC[9:0]=10'b111110011;

HDRCOMP=1;
FIFOHDR=8'b00000100;

FIFOCOMP=1

48

4. BYTE Transfer Machine

The final FSM in the transfer hierarchy is the BYTE state machine. The function of this state machine is to transfer the header information and the 82 bytes of information from each FIFO in the array. Enabled by the assertion of any FIFOENG signal, the state machine transitions from the BYTSTRT state. Transitioning to state BYTE2, the FSM asserts CUARTCSN and CTRLADDR identifying to the UART the specific register to write the data. State BYTE3 next asserts the address strobe signal, CUARADSN, and forms the header on the CTRLDATA output by OR'ing the contents of the registers CHDR and FHDR. This state additionally asserts the signal Header Enable (HDREN), which allows the FSM to drive the bi-directional CTRLDATA bus output lines. Transitioning to state BYT4, the state machine asserts Control UART Read Enable (CURDEN) allowing the UART to read in data on the CTRLDATA bus. The state machine transitions to state BYTE5 setting the signal BYTCNT equal to one. This signal is used to control the number of transfers that the BYTE machine conducts. State BYTE7 deasserts the output enable signal, HDREN, allowing the FIFOs to drive the CTRLDATA bus. This state also asserts the signal HDONE indicating to the FIFOXFER machine that the header information has been transferred allowing it to assert the FIFOCTRL lines. The FSM then transitions through states BTYE7 through BYTE10. These states act as a 'FOR' loop cycling the data out of each FIFO by asserting and deasserting FIFO Write Clock (FWRCLK) signal. Additionally, as the FSM cycles between these states the register Byte Count (BYTCNT) is incremented. When the BYTCNT is equal to eighty-two, the state machine transitions to state BYTE11 asserting the FIFOCOMP signal and

the FIFOCOMP signal and deasserting the HDRDONE signal. The state machine repeats

the process with the machine transitioning through the states until the FIFOENG signal is

deasserted.

Figure 4.11. BYTE Transfer State Machine

The completion of the system controller design finishes the design and

programming of the programmable logic devices. With the completion of these

programmable devices and installation in the TMR board, the only remaining change to

implement is the addition of white wires. The next chapter presents the addition of the white wire and required design changes.

THIS PAGE INTENTIONALLY LEFT BLANK

V. DESIGN COMPLETION

Any design when transitioned to a second engineer for completion is a difficult process. The new engineer not only has to have confidence in the design of his predecessor, but also still remain alert for potential problems. This chapter discusses the addition of required logic changes with the addition of white wires and the discovery and correction of a hardware incompatibility.

A. WHITE WIRES

The addition of the white wires to add additional logic to the TMR design was discussed in Reference [2]. The main concern of this author was the difficulty with the addition of these wires. The white wires required were primarily from the Xilinx FPGAs, which are 240 pin devices. The small scale of this device required a high level of skill to implement the changes.

The changes were completed with the assistance and expertise of David Rigmaiden of the Space Systems Academic Group. Utilizing a precise soldering station and microscope, the addition of the white wires took over a day and a half. The soldering of the pins to the FPGA required a high degree of precision and was very time intensive. With the addition of the white wires complete, the TMR R3081 system was ready for initial testing.

B. VOLTAGE REGULATOR

As discussed earlier any transition of a design from one engineer to a follow on engineer is a wary process. Upon transition of the board from Capt. Summers, it was necessary to conduct a thorough review of the design. It was during this review and the

power/ground checks discussed in chapter VI that a device compatibility problem was found.

The Xilinx FPGA utilized in the design fabrication was the XC4013-XLA. These devices provide the voting function that is the cornerstone of the design. The FPGAs soldered to the board required a supply voltage of 3.3 Volts. This posed a problem since the TMR board contained a 5-Volt and 12-Volt bus. There were two possible solutions to this problem. The first was to purchase new 5-Volt FPGAs and replace the 3.3V FPGAs. The second possible solution was to create a 3.3V bus for the FPGAs.

The first solution though sounding simplistic was quite difficult. The 240 pin FPGA is a flat pack device precisely soldered to the board. The unsoldering of these devices would lead to their destruction. The difficulty then came of soldering the new devices to the board. The expertise and precision that was required to the soldered the devices is very labor intensive. The slightest error though would lead to faults in the board and would have dramatic effects on any troubleshooting.

The second solution, the addition of a 3-Volt bus to the board for the FPGAs sounded more promising. The only problem with this solution was the impact of noise on the FPGA.

It was decided to proceed with the addition of a 3-Volt bus to the TMR design. This solution posed the minimal amount of difficulty. Additionally, this was a low cost solution when compared with the cost of new FPGAs. The following sections discuss the selection of the Voltage Regulator and the addition of the 3V bus.

1. Voltage Regulator

The selection of a voltage regulator first required the calculation of the current requirement of the FPGAs. The Xilinx website details a formula that is dependent on the number of logic cells utilized in the design. [Ref. 13]

$$P_{INT} = Vcc * Kp * Fmax * N_{LC} * Tog_{LC}$$

Where: $Vcc = 3.3$ V; $Kp = 28x10^{-12;}$ $Fmax=25Mhz$; $N_{LC}=1300$; $Tog_{LC}=.20$

Kp is a constant, which depends on the logic family. N_{LC} is the number of logic cells used in the design. For the worst-case scenario, the maximum number of logic cells was utilized. Tog_{LC} is the average percentage of logic cells toggling at each clock. The Xilinx recommend a using 20 percent for this value.

The calculated power requirement for each FPGA was .6006 Watts. Then using the total wattage of 1.8 and the input voltage of 3.3 Volts, a current estimate of .546 Amps was calculated. Utilizing this information, the Maxim 832 voltage regulator was selected. This fixed 3-Volt voltage regulator has an 8 V to 30 Volt input ranges and is rated at 1 Amp. Now with the device selected, a design to add the 3-Volt bus to the TMR system was required.

2. 3-Volt Bus

The addition of a three-Volt bus to the three FPGAs on the TMR board required a high level of skill and expertise. Each FPGA had 16 VCC pins. Additionally, each FPGA had a number of capacitors. The first step in the process was to unsolder a pin on

55

an FPGA and discover the amount of it would permit. Keep in mind that all of this work is accomplished with a microscope. The first pin on the system controller FPGA when unsolder from the board would only permit about a 20 degree bend. The pins on the FPGA are extremely delicate and the danger of reaching the breaking point was high. The remaining pins on one side of the FPGA were unsolder and lifted to the same angle.

The next step was to solder a wire to create the 3V bus. The first step consisted of laying a small strip of cellophane tape over the remaining soldered FPGA pins for protection. The distance was measured between the lifted pins to cut small tubing for protection. A wire was then solder to the first pin and the small tubing slid on. This process was repeated around the FPGA creating a 3-Volt bus. The remaining step was to unsolder the capacitor leads and solder them to the bus. The completion of this wired bus around the each FPGA led to the final step the addition of the voltage regulator board.

In order to make the addition of the 3-Volt bus to the board simplistic and permanent in nature it was decided to use an evaluation kit provided by Maxim. This kit provides a regulated 5.0V output voltage and is fully tested on a surface-mount printed circuit board. The kit comes with a MAX831 IC, but was easily converted to the use the MAX832 (3.3V output). After soldering the MAX832 onto the board, it was tested to confirm a constant 3.3 Volt 1 Amp output. This small printed circuit board was then permanently mounted over the FIFOs on the TMR board. The output from the board was then connected to buses surrounding the FPGAs. The photograph in Figure 5.1 depicts the work and components discussed above.
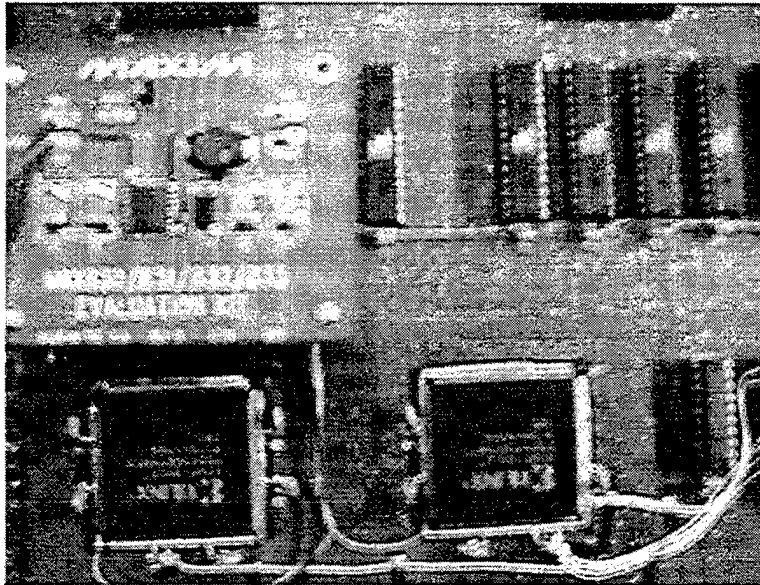
Figure 5.1. MAXIM Voltage Regulator

The addition of the voltage regulator marks the final hardware change

implemented to the design. This allowed transition to the next phase of the project,

design testing. The testing phase begins with basic power and ground checks, which is

presented in the next chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. SYSTEM TESTING

When testing a new programmable system one cannot be certain whether a specific problem or "bug" is caused by the hardware being faulty or if the software is incorrect. The objective of the testing is to identity any cause for a malfunction in the shortest possible period of time. Since there are numerous places in the system where problems may exist, the process has to be as efficient as possible. One efficient method of testing is to separate it into phases to ensure basic operation before proceeding to more complete functions. This method was utilized in the testing of the TMR board and it is broken down into the following two phases of initial and system level checks.

A.   INITIAL CHECKS

Before attempting to operate the system, initial checks of the board were conducted. These initial checks consisted of three steps: power/ground, clock signal and reset signal testing. The purpose of this initial testing is to ensure safe operation of the board and connectivity to system devices. Though these tests may seem simple and mundane, the bypassing of these checks could lead to unnecessary troubleshooting in later testing phases or damage to components.

1. Power Ground Testing

Before powering the board, it is essential that the verification of the connection of power and ground be verified. This procedure verifies that no manufacture defects are present in the board from fabrication. The chance of this occurring is slight as a result of the automation used in the fabrication process, but the potential impacts are very crucial. Bypassing this step could allow a shorted pin to destroy board components or lead to

unnecessary errors. This procedure was conducted utilizing a digital multi-meter. All component level power and ground pins were verified and no shorts were found. It was during this phase of the testing that the voltage compatibility problem of the FPGAs was discovered and verified. The FPGAs utilized in the board fabrication were low voltage devices, while the board was designed with a 5-volt bus. The solution to this problem was discussed in chapter V. The completion of these tests certified that the board was safe to apply power to and enabled movement to the next phase of testing the clock signals.

2. Clock signal testing

Another input that has to be verified that affects all components is the clock input. The clock waveform has to be correct in terms of voltage and timing. If the clock inputs are not correct, they have to be repaired before proceeding to the next step. The IDT3081 does not have a definite time relationship between the input clock and the SYSCLK output signal. The IDT manual contains a clock synchronization algorithm to conduct upon powering up the board, which can be found on page 11-8 of Reference 12. This process is applicable to the R3081 multiple processor designs utilizing ½ frequency bus mode. This procedure consists of performing a normal reset and selecting full frequency bus mode. This will force all processors to align the phases of their output clocks. Next, allow reset to be de-asserted for at least two to three clock cycles and then assert it. Now select one-half frequency bus mode and de-assert reset.

Utilizing an Oscilloscope, initial test of the TMR board verified an 11 MHz signal for the UART and an intermittent 20 Mhz clock signal for the 3081. The 20 MHz clock

signal is inputted to the microprocessor an output as a 10 MHz clock signal. The measured clock input to the 3081 was intermittent and tracing the difficulty led to the oscillator. Initially a failed oscillator was suspected. Further troubleshooting determined the oscillator had a bent connector that was not allowing proper seating. Upon reseating, the clock-input signal to the 3081 was reliable.

3. Reset Signal

The last signal to verify in the initial testing is the reset logic. The reset signal allows for the FPGA programming to be completed from the synchronous PROMs prior to the processor attempting to make a memory access. This time period allows the FPGA to map the voting and address control logic. If this did not occur in the correct logic sequence, the processor would experience a bus error.

The logic of the reset sequence was tested with an oscilloscope. The oscilloscope was connected to the reset pin of the 3081 processor. Then by asserting the board or system level reset push button the assertion of the reset signal was observed verifying correct operation. The time delay for the reset of the FPGA is unobservable. This step completes the initial testing phase and allowed us to proceed to the system level testing.

B.   SYSTEM LEVEL TESTING

By completing these initial electrical checks first, the scope of possible trouble within the overall system was minimized. The next checks confirm proper operation of the basic system components, the microprocessor, memory, and UART. The microprocessor has to be capable of reading and writing data to and from memory before

it is possible to proceed to further testing. The first step in this process is the programming of the microprocessor.

1. Processor Initialization

The IDT3081 is a 32-bit RISC microprocessor that is designed using the MIPS architecture [Ref. 12]. Although most programming occurs using high-level language, like "C", the programming during the testing was written in assembly language. This enabled precise control and setup of the processor structure by directly initializing the control registers. The following sections will first discuss the operations necessary to initialize the 3081 to a functional state and then the operational purpose of the program. Keep in mind, the program discussed in the following sections and given in Appendix C sets the processor in a known state for uncached operations only.

Before discussing the execution of the written program, this section will impart an understanding of the process of creating a boot program. The program is written in assembly language programs using a general word processor such as Notepad. This file is then assembled and linked using the Generic Cross Compiler (GCC) and linker provided by IDT. A makefile, which is similar to batch file, is utilized to assemble and link the file automatically. This makefile is listed in Appendix C and details options used by the compiler and linker. One important fact of information is the location that the code is assembled to start at in memory, which is at address 0x1FC00000. After a hardware reset, code will be running is KSEG1 (uncached) which contains address 0x1FC00000. When the processor is setup in kernel mode, four kernel/user segments (KUSEG) memory spaces are available. KSEG1 is a 512 Mbytes segment that is used for I/O

62

registers and boot ROM code. The address 0x1FC00000 corresponds to the processor's reset vector. The processor retrieves the first instruction here and begins program execution on power- up. When the compiler and linker have assembled the file, they output a memory map file, a binary file, and a Motorola s-record file that is downloaded into ROM. The S-record file is downloaded using a computer into a device programmer.

The next step in the process is to prepare the ROM. The ROM utilized in the TMR design is the AMD 27C010 128kx8 UV erasable. The first step in the burning process is to either erase the devices or verify they are blank. This is accomplished by placing them under a UV lamp for a period of 20 to 40 minutes erases the ROM. The time period is determined by the life of the devices. The more erasures a device has experienced the longer the period of time to erase them. Once the PROMs are erased and tested by the device programmer as blank, they are ready to burn.

The final step in the process is to determine if the file has to be split. This is determined from the size of the ROM used in the design. For example, the TMR board is using four 128 x 8 PROMs and each PROM has eight data output lines. Therefore, the file is split twice. The first 512K splits the memory in half and the next 512K split separates the halves again. The result is four separate data blocks that are programmed in the following address ranges 0-1FFFF, 2-3FFFF, 4-5FFFF, and 6-7FFFF. The first PROM outputs data on lines D0-D7, the second D8-D15, the third D16-D23, and the fourth D24-D31.

a) Status Register

Upon reset or initialization, the IDT 3081 boots up in an unknown state. The programmer's first priority is to force the processor into a known state by writing to the status register. This register controls the setup of the processor and memory management functions. The 32-bit register format is given in Figure 6.1.

31	28 27 26	25 24 23 22	21	20	19	18	17	16 15	8 7 6	5	4	3	2	1	0
CU (Cu3...Cu0)		RE	BEV	TS	PE	CM	PZ	SwC	IsC	IntMask Intr5..0, Sw1:0		KUo	IEo	KUp	IEp	KUc	IEc
4	2	1 2	1	1	1	1	1	1	1	8	2	1	1	1	1	1

CU: COPROCESSOR USABILITY
BEV: BOOTSTRAP EXCEPTION VECTOR
TS: TLB SHUTDOWN
PE: PARITY ERROR
CM: CACHE MISS
PZ: PARITY ZERO
SwC: SWAP CACHES
IsC: ISOLATE CACHE
RE: REVERSE ENDIANNESS

IntMASK: INTERRUPT MASK
KUo: KERNEL/USER MODE, OLD
IEo: INTERRUPT ENABLE, OLD
KUp: KERNEL/USER MODE, PREVIOUS
IEp: INTERRUPT ENABLE, PREVIOUS
KUc: KERNEL/USER MODE, CURRENT
IEc: INTERRUPT ENABLE, CURRENT
0: RESERVED: READ AS ZERO
MUST BE WRITTEN AS ZERO

Figure 6.1. Status Register Format. From Ref. [12]

Only a few of the bits in this register are defined after hardware reset initialization which are: the CPU is in kernel mode, KUc=0, interrupts are disabled, Iec=0, the data and address caches are not swapped, SWc=0, and the processor is in bootstrap mode, BEV=1. The initialization command used in the program OR'd three global variables SR_PE, SR-CU0 and SR_BEV. These global variables are defined in a header file that is linked with the main program. The three variables represent the options chosen for the boot up and are required for proper operation. This command is written to the status register which resets the parity error, set Coprocessor one useable, Coprocessor zero to kernel mode, clears interrupt masks, and set the processor in bootstrap mode. The assertion of the Bootstrap exception vector (BEV), determines the location of the exception vectors of the processor. The boot code utilizes uncached space, therefore the

64

BEV is set to 1, and the exception vectors reside in unchacheable space versus cacheable space.

b) The Cause Register

The cause register is a 32-bit register that describes the last exception conducted by the processor. The register's format is given in Figure 6.2. With the exception of the Software interrupt or SW bits, all other bits in this register are read only. The SW bits can be utilized to force an interrupt pending signal to the processor or clear a pending interrupt by writing a "0". This last step, clearing the interrupts, is required in initialization of the 3081. In the program, the global variable C0_CAUSE was written to the register to clear software interrupts. Further information on any of the fields in the Status or Cause register not discussed above can be found in Reference [13].


```
 31                                                              0
┌──┬──┬──┬────┬────────┬────┬──┬──────────┬──┐
│BD│ 0│CE│  0 │ IP[5..0]│ Sw │ 0│ ExcCode  │ 0│
└──┴──┴──┴────┴────────┴────┴──┴──────────┴──┘
  1   1   2   12     6      2   1      5      2
```

BD: BRANCH DELAY   ExcCode: EXCEPTION CODE FIELD
CE: COPROCESSOR ERROR
IP: INTERRUPTS PENDING   [0] : RESERVED
Sw: SOFTWARE INTERRUPTS*   Must Be Written as 0
                                        Returns 0 when Read

*READ AND WRITE. THE REST ARE READ-ONLY.
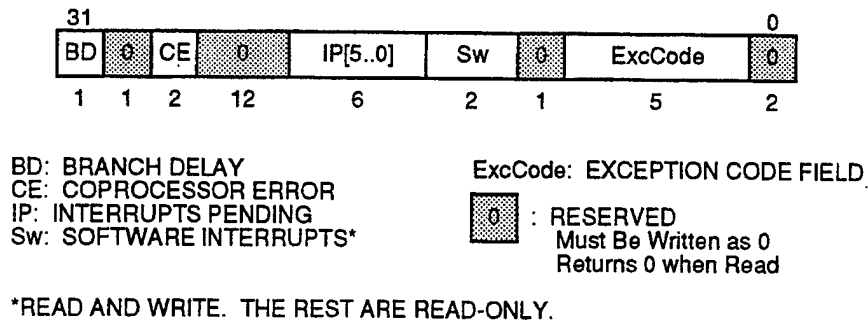
Figure 6.2. Cause Register Format. From Ref. [12]

c) Functional Testing

One of the most difficult problems with verifying the functionality of a system is the possibility of software errors. If the software does not properly setup the processor or function in the designed manner then the overall testing of the system is

worthless. This was not the case in with the testing of the TMR board. In the search for research information, the University of California at Davis was an exceptional source. Their experience with the ARGOS satellite project on Fault Tolerance software design and the IDT 3081 was a source outstanding resource. An application Engineer, Mr. Lance Halstead, at the University was also interested in building programs for the 3081. Providing him with copies of a program designed and developed for the TMR board, he made minor modifications for his specific board to test the program. The program functioned correctly by outputting a signal to LEDs and counter on his board. The verification of the code limits functionality problems to the board hardware logic. This is a major step in the design process, because problems are limited to the board hardware design. The program verification by an outside source on an operational design eliminates the possibility of software errors.

C.   TEST DATA AND WAVEFORMS

This section will discuss the data and waveforms measured and recorded utilizing the HP 16500B digital logic signal analyzer. The logic analyzer is utilized to record data on signal lines throughout the TMR design. The only limitation placed on the number of signals analyzed at one period was the number of data pods available. Each data pod connects to the back of the logic analyzer and has 15 input lines for a data signal. One pod signal line is connected to the system or master clock. After all connections to the board are complete, the logic analyzer is setup with labels corresponding to the data lines.

The logic analyzer is next set in one of two modes timing or state analysis. Timing analysis acquires data and stores it at equal time intervals. The internal clock of

the logic analyzer controls the time interval. State analysis is acquires data and stores it while a system is under test. The main difference between the two is that the clock for the state mode is supplied by the system under test.

The final step is to set a trigger for the logic analyzer to record data on. The trigger is selected by reviewing the output from the compilation of the assembly code. For example, in the ROM assembly language program, the first instruction in the code is 0B F0 00 62. This is the opcode for a jump instruction, which is 0000 10 in bits 31-26. The jump address is formed by taking bits 31-28 (0xB) from the PC and using the offset found in bits 25-0 of the instruction, shifting left by two to give bits 27-0 of the address. Thus for 0B F0 00 62, the offset is 11 1111 0000 0000 0000 0110 0010 and shifting left by 2 gives 1111 1100 0000 0000 0001 1000 1000 or 0xFC00188. This is correct code for a "jump start", where the start label is at BFC0 0188. Triggering on this instruction obtained the setup of the processor and the data on continuous read/write loop.

1. ROM Read

The following data segment is only a small portion of the data captured with the logic analyzer. The board was setup with the initial test program that read in the program code form the ROM and executed a write cycle of 0x00000000 to the KESG0 memory area. The purpose of the code was to verify the TMR board was reading program code correctly from the ROM.

Following the data segment in Table 6.1, the processors start the bus cycle with the assertion of the read, RD, signal in label 1516. The A/D bus is driven with the voted address 0xAD000000 and latched in with the assertion of ALE in label 1516. In label

67

1517, ALE is negated allowing the A/D bus to be driven by the voted data 0x23080004.

The memory control asserts is signal by the assertion of the Voted read signal in label

1517 to assert the read enable, RDEN, and read data enable signals. A single wait state is

inserted in the cycle because of EPROM data delay. The cycle is ended with the negating

of the read signal in label 1521. A waveform of the data captured by the logic analyzer of

the read sequence data is given in Figure 6.3.

Label Base	> >	AD31.0 Hex	ALE Bin	RST Bin	INT3.5 Binary	RD Bi	VOTRD Binary	BUSERN Binary	ACKN Bina	DATAEN Binary
1515		AD000000	0	1	100	1	1	1	1	1
1516		AD000000	1	1	100	0	1	1	1	1
1517		1FC001A0	0	1	100	0	0	1	1	0
1518		AD000000	0	1	100	0	0	1	1	0
1519		25080004	0	1	100	0	0	1	1	0
1520		25080004	0	1	100	0	0	1	1	0
1521		25080004	0	1	100	1	1	1	1	1
1522		25080004	0	1	100	1	1	1	1	1
1523		25080004	0	1	100	1	1	1	1	1
1524		25080004	0	1	100	1	1	1	1	1
1525		25080004	0	1	100	1	1	1	1	1
1526		25080004	1	1	100	1	1	1	1	1
1527		00001050	0	1	100	1	1	1	0	1
1528		00000000	0	1	100	1	1	1	0	1
1529		00000000	0	1	100	1	1	1	1	1

Table 6.1.A. ROM Data List

68

Label Base	>	DATAEN Binary	BUSREQ Binary	RDCEN Binar	DIAG10 Binary	MEMEN Hex	MEMCTL Binary	SYSCLK Hex
1515		1	1	1	111	A0	0110111011110	0
1516		1	1	1	110	A0	0110111011111	1
1517		0	1	1	111	A0	0110110011110	0
1518		0	1	1	111	01	0110110011111	1
1519		0	1	0	111	01	0110110011110	0
1520		0	1	0	111	00	0110010011111	1
1521		1	1	1	111	00	0111010011110	0
1522		1	1	1	111	A0	0110111011111	1
1523		1	0	1	110	A0	0110111011110	0
1524		1	1	1	100	A0	0110111011111	1
1525		1	0	1	100	A0	0110111011110	0
1526		1	0	1	110	A0	0110111011111	1
1527		1	0	1	100	A0	0110101111010	0
1528		1	0	1	100	E1	0010001111011	1
1529		1	0	1	100	E1	0011001111010	0

Table 6.1.B. ROM Data List

Label Base	>	SYSCLK Hex	VOTADD Hex	WRITE Hex	RDATAE Hex	RDENN Hex	CYCEND Hex	RDCENN Hex	EPROMCS Hex
1515		0	006B	1	1	1	0	1	0
1516		1	006B	1	1	1	0	1	0
1517		0	006B	1	1	1	0	1	0
1518		1	006B	1	0	0	1	1	0
1519		0	006B	1	0	0	1	1	0
1520		1	006B	1	0	0	0	0	0
1521		0	006B	1	0	0	0	0	0
1522		1	006B	1	1	1	0	1	0
1523		0	0068	1	1	1	0	1	0
1524		1	0069	1	1	1	0	1	0
1525		0	0068	1	1	1	0	1	0
1526		1	0416	0	1	1	0	1	1
1527		0	0416	0	1	1	0	1	1
1528		1	0416	0	1	1	1	1	1
1529		0	0416	1	1	1	1	1	1

Table 6.1.C. ROM Data List
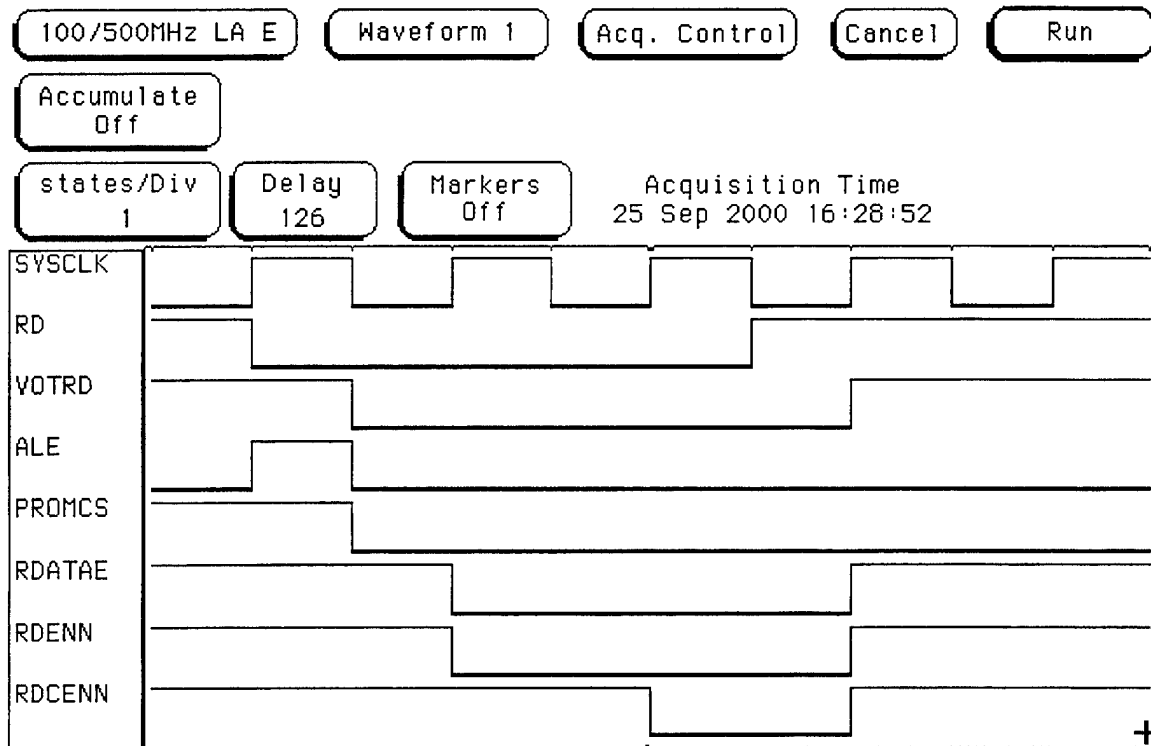
Figure 6.3. ROM Data Waveform

2. RAM Write

The purpose of this program was to verify the ram data write cycle. A RAM read was not possible, because the RAM segment is located in a cache memory section. The initialization of the cache was not necessary for this program execution. This program's function was write the data word 0xDEADBEEF to an address in the RAM segment.

The data sequence in Table 6.2 details the signals during a RAM write operation. In label 128, the processors have asserted the Write, ALE, RAMCS*, and the A/D bus with the address 0x000000000. The processor next negates ALE, latching in the address. In label 129, the A/D bus is now driven with the data 0xDEADBEEF. The assertion of the write signal and the majority voting leads to the assertion of the voted write signal,

70

VOTWR. The memory controller PLD responds to this by asserting the write data enable, WRDAEN, and acknowledge, ACKN, signals in label 129 and 130. The processor ends the bus cycle by deasserting write in label 131. A waveform is given with respect to this process and data segment in Figure 6.4.

The data segment in Table 6.2 also confirms the TMR FPGAs executing the majority voting properly. During the code execution and as seen below, the INTCS* signal remained deasserted. The correct execution of this program verifies the transfer of data between components on the board. The next test step is data output from the UART, which is presented in the following section.

Label Base	> >	WRITE* Hex	VOTEWR* Hex	RDEN* Hex	CYCEND* Hex	RDCEN* Hex	RAMCS* Hex	ROM1 Hex	INTCS* Hex
127		1	1	1	0	1	1	00	1
128		0	0	1	1	1	0	00	1
129		0	0	1	1	1	0	00	1
130		0	0	1	0	1	0	00	1
131		1	1	1	0	1	1	00	1
132		1	1	0	1	1	1	00	1
133		1	1	0	1	1	1	00	1
134		1	1	0	0	0	1	00	1
135		1	1	0	0	0	1	00	1
136		1	1	1	0	1	1	00	1
137		1	1	1	0	1	1	00	1

Table 6.2.A. RAM Data List

Label Base	>	BUSREQ Binary	RDCEN Binar	DIAG10 Binary	MEMEN Hex	MEMCTL Binary	SYSCLK Hex
127		0	1	110	A0	011010111110	0
128		0	1	111	E1	001000111111	1
129		0	1	110	E1	001100111110	0
130		0	1	110	A0	011011111111	1
131		0	1	110	A0	011011001110	0
132		0	1	110	01	011011001111	1
133		0	1	110	01	011011001110	0
134		0	1	110	00	011001001111	1
135		0	0	110	00	011101001110	0
136		0	0	110	A0	011011101111	1
137		0	1	110	A0	011011101110	0

Table 6.2.B. RAM Data List

Label Base	>	AD31.0 Hex	ALE Bin	RST Bin	INT3.5 Binary	RD* Bi	BUSER* Binary	ACK* Bina	DATAE* Binary
127		25080004	0	1	100	1	1	· 1	1
128		00000000	1	1	100	1	1	1	1
129		DEADBEEF	0	1	100	1	1	0	1
130		DEADBEEF	0	1	100	1	1	0	1
131		DEADBEEF	0	1	100	1	1	1	1
132		1FC001B0	1	1	100	0	1	1	1
133		DEADBFEF	0	1	100	0	1	1	0
134		24019FFF	0	1	100	0	1	1	0
135		24019FFF	0	1	100	0	1	1	0
136		24019FFF	0	1	100	0	1	1	1
137		24019FFF	0	1	100	1	1	1	1

Table 6.2.C. RAM Data List
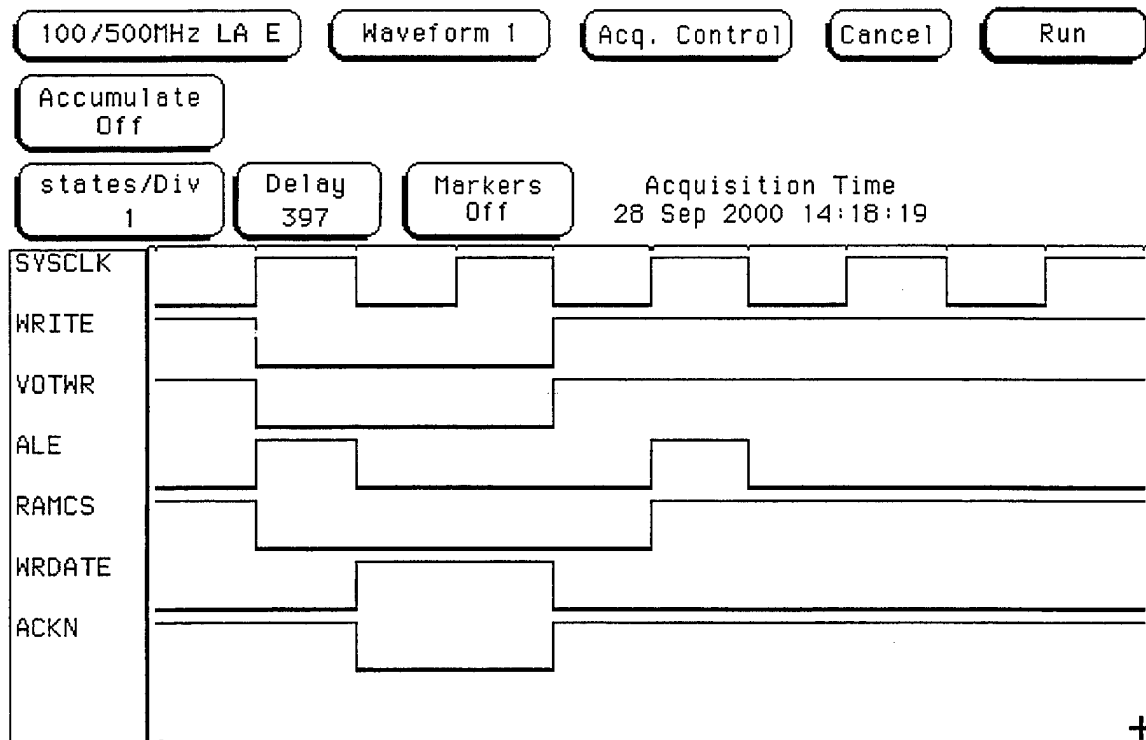
Figure 6.4. RAM Data Waveform

3. UART DATA

The function of the UART program is to initialize the UART for I/O and write a continuous data stream to the transmit register. This program, listed in Appendix C, initializes the processor in the exact same manner as the two previous programs but the remainder of the program is entirely different. The program first initializes the 16550 UART by writing data to the line control, FIFO control, modem control, and divisor latch registers. This initializes the UART for eight data bits, no parity, and in polling mode. Next, 0x83 or 72 decimal is written to the divisor latch register for a baud rate of 9600. The program then enters a continuous write loop that outputs the data, 0x2322223, which represents the ASCII character # to the transmit register.

The data sequence in Table 6.3 details the signals during a UART write operation. In label 378, the processors have asserted the Write, ALE, and the A/D bus with the address 0x1FE00000. The processor next negates ALE, latching in the address. In label 379, the A/D bus is now driven with the data 0x23222223. The assertion of the write signal and the majority voting asserts the voted write signal, VOTWR. The memory controller PLD responds to this by asserting the write data enable, WRDAEN, and acknowledge, ACKN, signals. The processor ends the bus cycle by deasserting write in label 380. A waveform is given with respect to this process and data segment in Figure 6.5.

Additionally, the output data was verified by connecting a PC to the output port of the UART. The PC was running the HyperTerminal program and configured to match the setup characteristics of the UART. A continuous data stream of the ASCII character # was outputted to the screen. The next step in the test sequence is to receive data input from the PC, modify this data and output the modified data to the PC. This program and data is presented in the next section.

Label	>	SYSCLK	AD31.0	ADDR32	WRITE	VOTWR	ALE	RD	VOTRD
Base	>	Hex	Hex	Binary	Binar	Hex	Bin	Bi	Hex
378		1	1FE00000	00	0	0	1	1	1
379		0	23222223	00	0	0	0	1	1
380		1	23222223	00	1	0	0	1	1

Table 6.3.A. UART Data List

Label	>	VOTBE	UARTCS	WREN	WRDAEN	ACKN	BURST	RDCEN	TIMRCS
Base	>	Binary	Hex	Hex	Hex	Bin	Bin	Bin	Hex
378		0000	0	F	0	1	1	1	1
379		0000	0	0	1	0	1	1	1
380		0000	0	0	1	0	1	1	1

Table 6.3.B. UART Data List

74

Label	>	RAMCS	UARTCS	COVTER	CYCEND	INT5.3	AVOTER	COVTER
Base	>	Hex	Hex	Hex	Binary	Binary	Binary	Hex
378		1	0	0	1	100	0	0
379		1	0	0	0	100	0	0
380		1	0	1	0	100	1	1

Table 6.3.C. UART Data List

Label	>	VOTINT	BUSERR
Base	>	Hex	Binary
378		0	1
379		0	1
380		0	1

Table 6.3.D. UART Data List
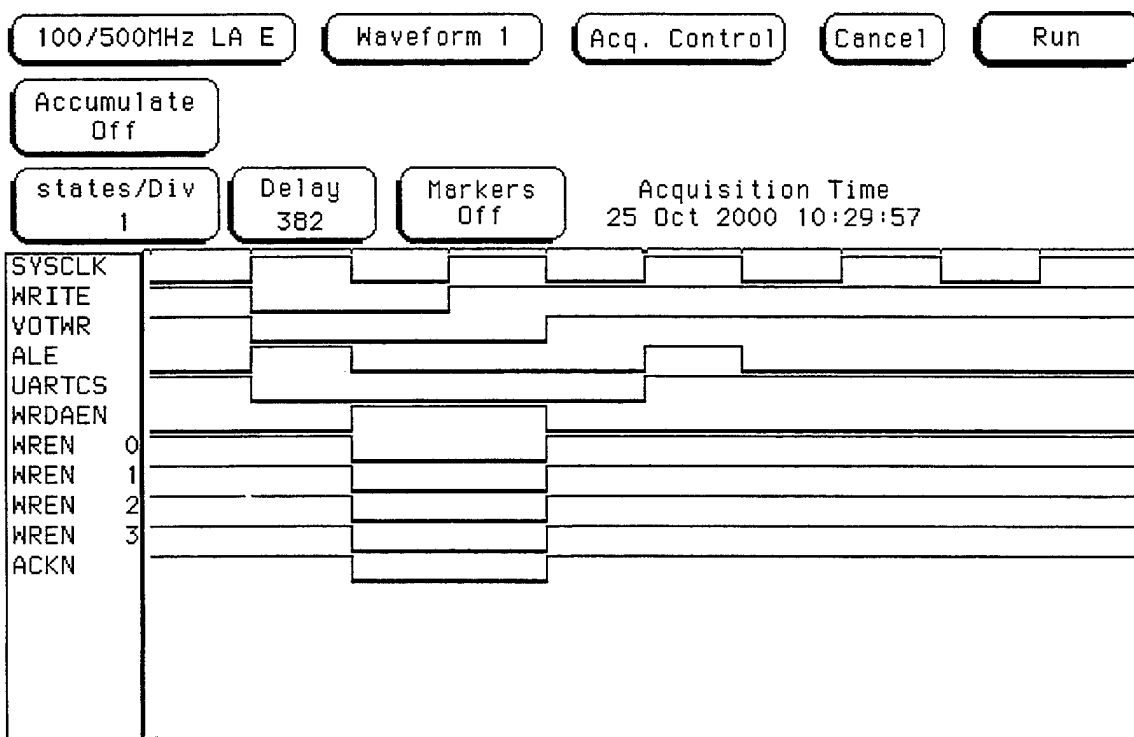
Figure 6.5. UART Data Waveform

4. UART INPUT/OUTPUT

The purpose of the UART I/O program was to test the board read cycle and storage of data input. This program read in data from the UART receive register, added 30 to this data, and transmitted this data back to the PC. This program differed from the previous UART code in the setup of the UART. Specifically, the modem control register was loaded with 0x23, which set the UART in Autoflow control mode. The Autoflow control mode enabled to UART to control the Request to Send (RTS) and Clear to Send (CTS) signals. This change was implemented to enable the program to output individual characters.

The program indicates it is running by outputting the ACSII character # to the PC. Then entered a loop, which checked the status register receive data bit. The assertion of this bit informed the UART that the receive register held one byte. The data that was read in is stored in the T7 register and 30 added to this quantity. The addition of 30 changed the uppercase ASCII character to a lower case character. The contents of register T7 is then transmitted to the PC.

The data sequence in Table 6.4 details the signals during the read in and output of the data. In label 18, the processors have asserted the Read, ALE, UARTCS*, and the A/D bus with the receive register address 0x1FE00000. The processor next negates ALE, latching in the address. In label 19, the A/D bus is now driven with the input data 0x00000054. This data is then stored in the register T7.

Label	>	AD31.0	ADDR32	ALE	SYSCLK	BUSERN	RD	VOTRD	RDEN
Base	>	Hex	Binary	Bin	Decima	Binary	Bi	Hex	Hex
18		1FE00000	00	1	1	1	0	0	0
19		00000054	00	0	0	1	0	0	0
20		00000054	00	0	1	1	0	0	0
21		00000000	00	0	0	1	1	1	1

Table 6.4.A. UART Receive Data

Label	>	WRITE	WREN	VOTWR	VOTRD	WREN	ACKN	CYCEND	INT5.3
Base	>	Binar	Hex	Hex	Hex	Hex	Bina	Binary	Binary
18		1	0	0	0	0	1	0	100
19		1	F	1	0	F	1	0	100
20		1	F	0	0	F	1	0	100
21		1	F	1	1	F	1	1	100

Table 6.4.B. UART Receive Data

Label	>	TIMRCS	EPROCS	RAMCS	UARTCS	COVTER	AVOTER	VOTINT
Base	>	Hex	Hex	Hex	Hex	Hex	Binary	Hex
18		1	1	1	0	0	0	0
19		1	1	1	0	0	0	0
20		1	1	1	0	0	0	0
21		1	1	1	0	0	0	0

Table 6.4.C. UART Receive Data

The data sequence in Table 6.5 details the signals during the output of the modified data. In label 194, the processors have asserted the Write, ALE, UARTCS*, and the A/D bus with the transmit register address 0x1FE00000. The processor next negates ALE, latching in the address. In label 195, the A/D bus is now driven with the modified input data 0x00000074. This data is written to the UART, which then outputs the data on the serial output port.

Label >	AD31.0 Hex	ADDR32 Binary	ALE Bin	SYSCLK Decima	BUSERN Binary	RD Bi	VOTRD Hex	RDEN Hex
Base >
194	1FE00000	00	1	1	1	1	1	1
195	00000074	00	0	0	1	1	1	1
196	00000074	00	0	1	1	1	1	1
197	00000200	00	0	0	1	1	1	1

Table 6.5.A. UART Transmit Data

Label >	WRITE Binar	WREN Hex	VOTWR Hex	VOTRD Hex	WREN Hex	ACKN Bina	CYCEND Binary	INT5.3 Binary
Base >
194	0	F	0	1	F	1	0	100
195	0	0	0	1	0	0	0	100
196	0	0	0	1	0	0	0	100
197	1	F	1	1	F	1	1	100

Table 6.5.B. UART Transmit Data

Label >	TIMRCS Hex	EPROCS Hex	RAMCS Hex	UARTCS Hex	COVTER Hex	AVOTER Binary	VOTINT Hex
Base >
194	1	1	1	0	0	0	0
195	1	1	1	0	0	0	0
196	1	1	1	0	0	0	0
197	1	1	1	0	0	0	0

Table 6.5.C. UART Transmit Data

This chapter presented the data captured during program execution. This data supports the implementation of the TMR design. The goal of the design is the utilization of COTS devices in space applications. In support of this effort, the next chapter discusses preliminary design considerations for a space flight board.

VII. CONVERSION TO SPACE FLIGHT BOARD

Once the testbed has been designed, implemented, and evaluated, the next logical step in the evolution of the TMR design is conversion to a space flight board. The conversion of the design depends on numerous factors, such as determination of parts to be replaced, size and power constraints just to name a few. The following sections will discuss issues that deal with the preparation of the TMR board for space applications. The focus of the design for the TMR board is with respect to the next satellite, NPSAT1, being designed and built by the Naval Postgraduate School. This satellite design constrains the power, weight and size for the TMR design.

A. CONSTRAINTS AND TRADEOFFS

Every satellite design conducts studies on the tradeoffs between different system designs. The following sections contain an overview of the design constraints and tradeoffs of the TMR design for a space flight board.

1. Power

Power is a precious commodity in space. A satellite is limited in the power that it has available by the sizes of the solar array and batteries. The TMR design inherently increases the power required compared to a normal processor board. The implementation of this design in space applications has to balance the performance and reliability against with the increased power consumption. The current system design utilizes low power consumption commercial parts. The conversion of this board to a flight ready design may require the use of radiation hardened (radhard) components. A radiation-hardened component is designed to continue functioning when exposed to high levels of radiation

over the lifetime of the device. For example, most radiation-hardened components are designed to continue operating to a total radiation dosage of 20 Krads. Thus, the device can sustain a dosage of four Krads per year for a period of 5 years. Though the TMR design is robust and does not allow errors to propagate past the processor, certain devices that implement the design require increased tolerance. For example, the FPGA contains the voting logic; these devices reliability has to be as close to 100 percent as possible. Their failure would cause a breakdown of the TMR design and lead to a total system loss. The decision on which parts will be radhard will depend on the role of the device, the power capability of the spacecraft and the life of the satellite. Radhard components generally require more power than their non-hardened counterparts do. The TMR design with non-radhard devices currently has power consumption between 6.5 to 9.5 Watts depending on the function of the board. The predicted maximum power available for NPSAT1 is 22 Watts. These factors go into the decision model for selecting devices to utilize in the design.

2. Size

The small size of NPSAT1 limits the area available for the TMR designed board. The current size of the TMR board is 13x12 inches. In researching the constraints placed on designs used in the previous Petite Amateur Satellite (PANSAT), it was discovered this board was too large. Boards previously utilized in the PANSAT design were on average were 6x4 inches. The solution to this is to section the board into multiple smaller boards interconnected by bus connectors. In inspecting the current TMR design, two alternatives were thought of for sectioning the board. The first alternative is to group the

components on boards by like device. For example, one board would hold all three processors and another would hold all FIFOs. The second alternative is to section the design by breaking the TMR design into individual component boards. An example of this is that one board would contain an all the RAM/ROM. The second option is by far the most efficient choice for two reasons. These reasons are design production efficiency and testing. The design engineer would only need to design the three identical processor boards and one assorted component board. This option would aid in testing by allowing each processor board to function alone for testing.

Figure 7.1 depicts one possible configuration for the boards. The processors and latches would be on one card, the memory, ROM and RAM, on another card. The Oscillators or clocks, the interrupt circuit, and PLD logic would be placed together. The last card would contain all three FPGAs and the Serial EEPROM.
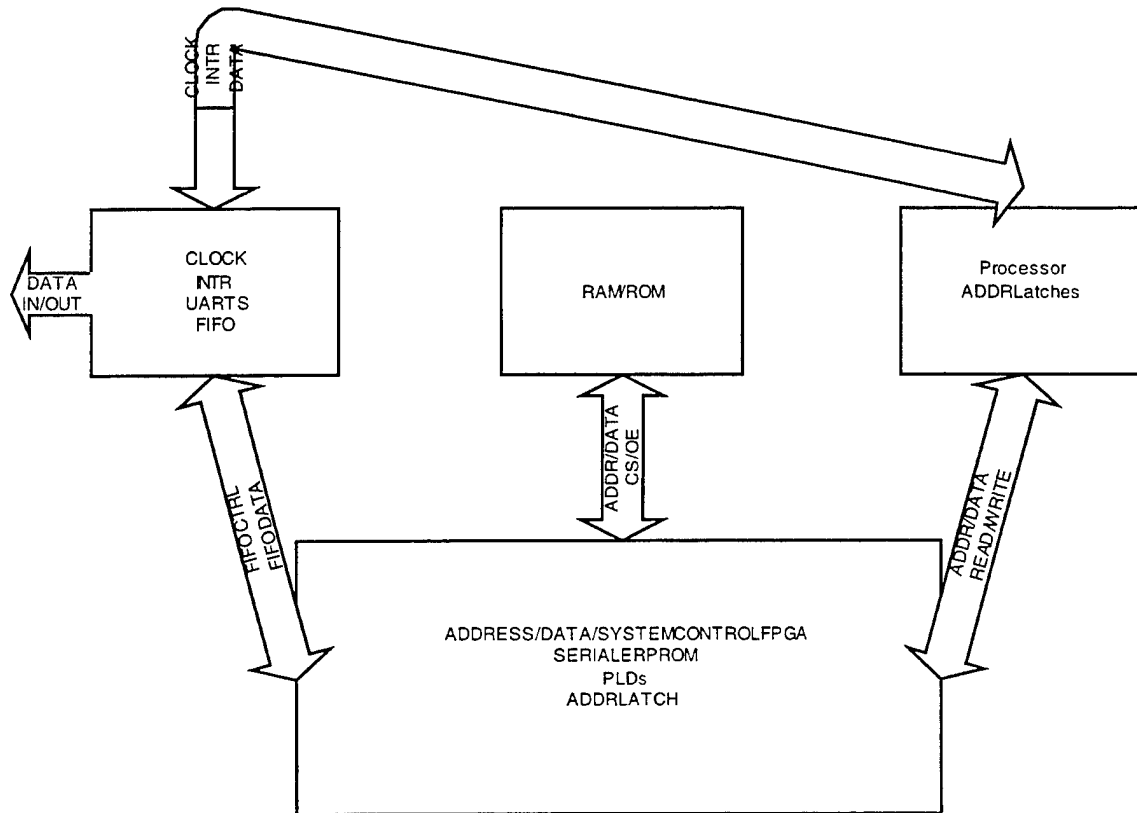
Figure 7.1. Space Design Layout

3. Vibration Analysis

A design to be utilized for space applications has to be tolerant of the severe shocks and vibrations experienced during launch. During launch into orbit, the body of the satellite experiences gravitational forces up to 9G. Once in space, the vehicle will the experience the shock of being released by an explosive bolt. Loss of connectivity as a result of such vibrations is avoidable with proper testing. The NPS Space Academic Group has the capability to conduct this testing dependent on the launch vehicle. In inspecting the board for possible problems, an immediate source of concern is the surface mounted FPGA. The FPGA are secured to board by only the soldered pins.

B.   SPACE FLIGHT PREPARATION

The preparation of a satellite for space is filled with numerous questions. When will the mission fly? Where will the mission fly? What systems are critical? What amount of shielding will be around the SEE devices? In addition, what is the mission life? All of these factors are critical in the selection of devices for a spacecraft and lead to a prediction of the radiation environment of a mission. The purpose of this section is to discuss some of the factors in the selection of components and design criteria.

1. Mission Parameters

As discussed in the background, the orbital parameters of a satellite decide the radiation environment it will experience. This in turn impacts numerous factors in the satellite design such as shielding thickness. The TMR system is destined to be a part of the next Naval Postgraduate School Satellite design. Preliminary orbital parameters for this satellite place it at an altitude of 800km, which is a Low Earth Orbit (LEO). The harshest radiation environment encountered by satellites in a LEO is the high-energy particles of the Van Allen Belt, which it will pass through numerous times each day. The energies of the protons in the belts can range from KeV to hundreds of MeV. The level of radiation flux experienced during of a satellite's life in the belt varies greatly with orbit inclination and altitude. Flux is the flow of energy per unit time and per unit cross-sectional area. At altitudes between 200 and 600-Km large increases in radiation flux levels are seen as the altitude increases. Above 600 Km, the radiation flux increases more gradually as altitude increases.

83

A LEO satellite, though subjected to the Van Allan belt, experiences a benefit from its lower altitude, which is somewhat protected from cosmic rays and solar flares by the earth's magnetic field. As the satellite altitude increases, the protection from these particles decreases. The amount of protection the satellite receives from the earth's magnetic field is also dependent on inclination. As the inclination of the satellite increases above 45 degrees, the satellite orbit enters the Polar Regions more frequently, which is beyond the magnetic field exposing it to the full effect of space radiation. Utilizing this information, a Radiation Effects Engineer is able to proceed to the next step in part selection, which is a radiation risk assessment.

2. Radiation Risk Assessment

A radiation risk assessment for any electronic components consists of the calculation of total dose damage and SEU susceptibility of the device caused by the predicted radiation environment of the spacecraft.

Knowing the orbit parameters allows the determination of the suitability of electronic devices in their intended application, dependent on the radiation environment to which the devices are subjected. Utilizing an altitude of 800 KM alone, a satellite will experience a predicted radiation dose (called ionizing dose) rate of approximately one krad (Si) per year. [Ref. 14]

An additional method available to predict the environment is the use of software radiation environmental simulations. There are multiple software programs available for the prediction of the SEE effects such as SpaceRad and Cosmic Ray Effects on Micro-Electronics (CRÈME96). SpaceRad is a commercial program and a demonstration

84

version was unavailable for evaluation. CREME96 is a free program available at the Naval Research Laboratory website. This program focuses on the prediction of the Single Event Effects and determining SEU rates, but to accomplish this requires radiation ground-test data on the devices. This software was utilized primarily to get a picture of the environment that the spacecraft would be exposed to as a prelude to assist in device selection.

CRÈME requires the user to input the orbit of the satellite and select the environment, either quiescent or extreme solar events. This information is entered in a User Request file. The orbital profile entered into the user request file an inclination of 30 degrees and at a perigee/apogee of 450 km. Figure 7.2 depicts fluxes of various elements, atomic number Z=1 (protons) to atomic number Z=8 (oxygen), vs. kinetic energy. For example, using atomic number 7 (nitrogen) and breaking this element into species based on kinetic energy. Then using the figure, the spacecraft would experience a flux of 10^{-6} from nitrogen elements with a kinetic energy of 1 MeV/nucleon. This allows an evaluation of the ionizing radiation environment of different elements at the external surface of the spacecraft for the orbital parameters specified. The recommendation for SEE calculations is to consider elements up to atomic number 92. This was not done due to the complexity of the figure. With this information, the user has the ability to determine the amount of shielding required to protect the spacecraft. This will give a jump off point for calculating SEU on different devices.
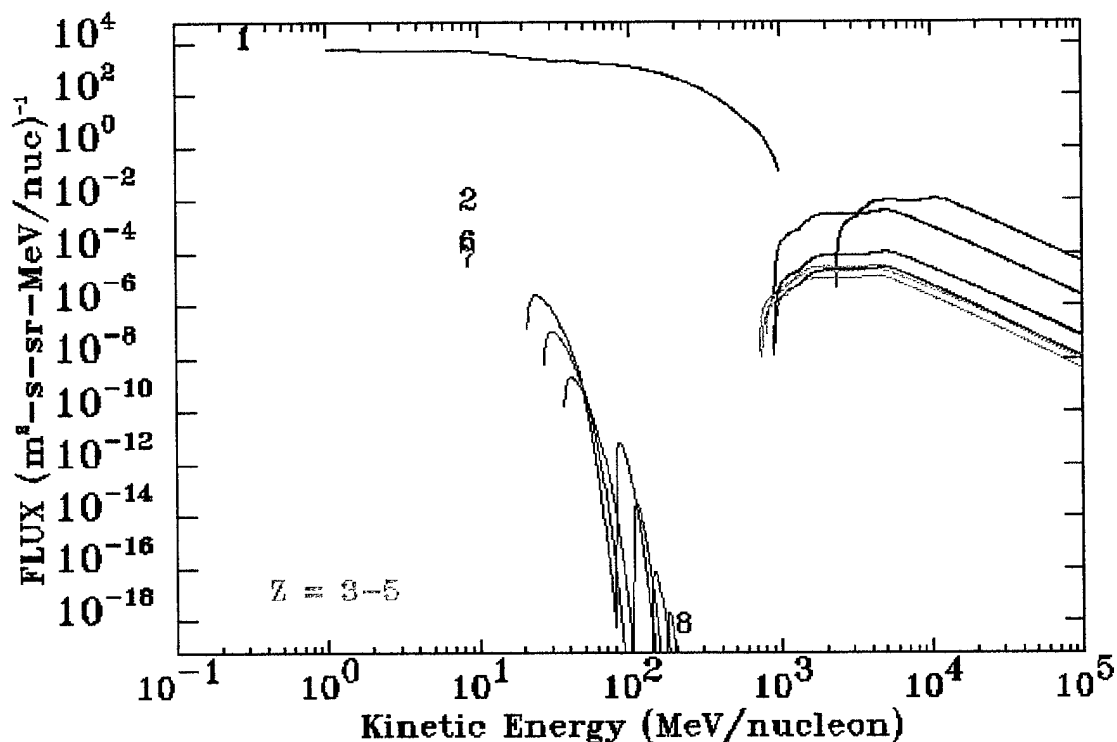
Figure 7.2. Differential Flux of various elements vs. Kinetic Energy at the external surface of the spacecraft Z=1 (protons) Z=8 (Oxygen)

3. Mission Specific

After mission planners and radiation effects engineers have planned the orbit configuration, date of launch, mission life, and nominal shield thickness, an analysis utilizing this data is performed. The first phase of this analysis is to decide on the SEU requirement determined by the function of the device. The role of the device in the system is critical to the analyses. The application of this process to the TMR system is vital. The following sections will discuss design approaches that can be used to increase radiation hardness and tolerance of the system to SEU.

a) *Microprocessor*

First and foremost in the design are the three IDT3081 microprocessors. The microprocessor is the brain of the entire design and its correct function is essential. Though, the TMR voting scheme protects the system from a propagating fault additional protection is necessary to increase system reliability. Previous radiation testing conducted on the IDT3081 concluded that the R3000 microprocessor could survive the proton SEU environment. [Ref. 15] Later testing of the IDT3081 predicted a proton upset rate per device of 1.4×10^{-4} upsets/day with 60 mils of aluminum shielding. [Ref. 16]

The function of the TMR system would be degraded considerably if a processor ceased functioning. In theory, if the other two processors still concur the system would continue functioning properly. However, if the two processors disagree and cause a voter interrupt, the system would be unable to determine the correct data. The implementation of a watchdog program in the operating system of the microprocessor is therefore essential. The watchdog timer is like an "I'm functioning" method of error control. If this message is not received within a certain period of time, a "time out" has occurred. The system will then perform some action such as a reset.

An additional method that can be implemented to increase system reliability is the filling of unused RAM with software interrupts instructions. These instructions would cause a graceful recovery if the processor jumped outside the programmed memory area.

b) XILINX FPGA

The next major device section in the TMR design is the voters and system controller FPGA. As stated earlier, the voter is the single point of failure in a TMR system. An error in these devices would lead to faults permeating throughout the system. The system controller performs the essential role of managing communication and transfer of information. A possible error here is not as detrimental as in the voters, but the communication function of this device is vital. The role of these devices in the design calls for any and all protection. Therefore, the use of radiation-hardened components is necessary and their use is seen as an essential cornerstone to the success of this design.

One of the key features of FPGAs is that they are reconfigurable. If a design change is necessary during flight, then a new configuration can be loaded and the functionality of the FPGA altered without having physical access. Unfortunately, this increased flexibility results in a more involved designed solution for SEU effects.

The straightforward approach to SEU mitigation is to reconfigure the FPGA upon detecting a system error or at specific time intervals. For example, an FPGA utilized to control a spacecraft thermal control system experiences an SEU, causing the FPGA to turn on the heater. If through onboard systems it can be determined the heater is on inadvertently, a command could be sent to reconfigure the FPGA.

A second approach that can be utilized is the readback function. This feature allows the reading of the state of every flip-flop and configuration cell within the FPGA. This is a capability provided by the Xilinx FPGA. This function runs in the background and does not affect the function of the FPGA. A CRC checksum calculated

from the readback bits is generated and inserted at the end of the bit stream. The checksum is then compared to the expected checksum for the current configuration, if it does not match, then an error has occurred. [Ref. 17]

c) Memory

The memory system is the next portion of the TMR design. This memory system discussion includes both RAM and ROM. The validity of memory in any computer system is of major concern. It stores and provides the data necessary for proper functioning. There are some protections that can be implemented to the memory system to increase reliability. The first would of course be an upgrade to a radiation-hardened component. Possible choices for replacement devices are given in a later section. The second is the addition of an Error Detection and Correction (EDAC) scheme such as parity error or hamming code, which can be added to the overall system design. The addition of EDAC can be accomplished in both software and hardware. The focus of the following will be on hardware EDAC. Keep in mind the following is discussed only to increase the overall reliability of the spacecraft. The TMR design is itself an Error Detection and Correction method. The voting modules detect the error by the majority vote and the error is corrected by only allowing the voted data pass.

EDAC is accomplished by the use of algorithms such as the Hamming Code. The Hamming code is suitable for detecting single errors in a code. It functions by encoding a block data with check code. The code is then check by the next unit and is able to detect the position of a single error and the existence of more than one error. By the determination of the position of the error, it is then possible to correct that error. The

level of EDAC protection desired determines the number of check word bits per data word. For example, providing single bit error correction with dual bit error detection to eight to fifteen data bit requires five check bits.

An EDAC IC functions by storing a check word in memory with the data word. During a read operation, the data word and corresponding check word are retrieved from memory. The EDAC IC generates a new check word determined by the data from memory and compares it to the check word stored in memory. If the two check words are identical, the data word is correct. If the check words are different, an error has been detected. Single-bit error corrections with dual-bit error detection ICs are available commercially. The placing of an EDAC component on the bus lines from the memory would force all data used by the microprocessor to pass through this device.

Another method of hardware EDAC that is quite common is parity. Parity, being a single bit at the end of a structure, simply indicates whether an odd number or an even number of ones was in that structure. This method detects an error if an odd number of bits are in error, but if an even number of errors occurs, the parity is still correct. For example, the parity is the same whether zero or two errors occur. Additionally, notice that this is a detect-only method of mitigation and does not attempt to correct the error. Therefore, an additional algorithm, such as Manchester-encoding, is also utilized. This encoding detects if the data is in the proper format. Parity generator/checker ICs are commercially available.

These EDAC methods, such as the Hamming code, are more efficient than TMR. They do have a downfall, which is they are very hardware intensive. For

90

example, the addition of an EDAC IC to the TMR design would introduce seven bits of additional data for each 32 bits of data. This would in turn require additional memory hardware components be added to the design to store this additional data.

d) Serial EEPROM

The serial EEPROM is in a separate category from the other memory devices. The EEPROM is utilized only in the programming of the FPGA. The programming of a device in space is problematic, increasing the chance of error in the program. As shown in the following section no radiation data was found on serial EEPROM. A small cross-section and susceptibility to SEU limit their use in space. The removal of this part and one time programming of the FPGA defeats the purpose of utilizing an FPGA.

An alternative that needs to be discussed is use of ASIC designs for a satellite. Programmable logic has the advantage over the ASIC in reduced cost and faster design time. The use of an ASIC also defeats the purpose of using a FPGA, the benefit of allowing on-orbit design changes. This flexibility allows a mission to adapt to changing requirements.

e) FIFO

The FIFOs in the TMR design are grouped into three banks of five for each microprocessor. The need for the FIFOs in a space flight board depends on the function of the board. If the desire is to later analyze data to discover the location of the fault then FIFOs are necessary. If on the other hand, the desire is to put a functional system board and not analyze the data, but just the performance of the system in response

to faults then the FIFOs can be removed. The FIFOs' only function is the collection and transfer of data after a vote interrupt. Therefore, the device performs an important but not essential analytical function. The current FIFOs are 1024X8, which is a common size for a bus. A design change that would improve reliability is the changing of these devices to 9-bit wide ICs. This ninth bit would enable the use of parity thus enabling detection of single bit errors.

f) Assorted

The remaining components on the board consist of address latches and buffer/drives. These components play an important role in any design. There is enough current data on the non-radiation hardened components provided in Appendix D that use is justified. The oscillators are the only remaining component not touched upon in the previous sections. The use of a radiation tolerant oscillator is not only recommended but also necessary for the design. The clock signal is the most important signal in the entire design. It synchronizes the voters and provides a reference signal to the processors.

C.   PART SELECTION

The next step in the transition process is to take the components list and search for radiation testing data on the devices. There are numerous Radiation Data Banks on the Internet for example ERRIC/DASIAC and the Goddard Space Flight Center both maintain radiation test lists. In searching for the devices or compatible devices to limit design changes, the information at these databases was lacking. None of the devices utilized in the design and very few comparable devices were found. The earliest database entries that were found on device testing is 1997 and that was on devices manufactured

92

two to three years previously. This information offered little assistance in device selection.

Next in the process was a search for radiation-hardened devices at companies such as UTMC and Honeywell. To qualify as a radiation-hardened device the manufacture has to guarantee operation at 20 krad. Hardness is simply a measure of the total dose of radiation to which an IC can be subjected before critical parameters cross a predefined threshold.

The designer has to keep in mind the most cost efficient method for meeting the SEU requirement is to make an appropriate combination of SEU-hard devices and other factors. The radiation devices though meeting mission parameters for radiation tolerance consume a large portion of energy. Appendix D gives a part breakdown of available radiation hardened compatible devices and their cost. It additionally lists suitable COTS devices and the radiation testing data available on the device.

THIS PAGE INTENTIONALLY LEFT BLANK

VIII. CONCLUSIONS AND FOLLOW-ON RESEARCH

The previous chapter provided preliminary design considerations for a space flight design. This chapter will present the conclusion drawn from the project and areas for follow-on work.

A. CONCLUSIONS

During this project, a fabricated TMR board design was modified, finished and tested. The design was expanded with the addition of a voltage regulator for the FPGAs. The UART was swapped for a newer version. The programming and addition of the system controller to the design enables the system to transfer data to the HCI. Burning programs into the ROM devices and capturing execution data on a digital logic analyzer tested the design. Finally, the research focused on the preliminary design requirements for a space flight board.

The data captured by the logic analyzer and presented in Chapter V demonstrated the conceptual and hardware implementation of the TMR concept to be valid. The measured waveforms agreed with the predicted waveforms presented in Reference 2. This research advanced the project by correcting and completing the hardware design, designing and executing test programs, and verifying the design worked as anticipated.

B. FOLLOW-ON RESEARCH

The advancement of this project one step further creates additional paths of follow-on research. This section discusses some of the areas where follow-on research is recommend.

1. Completion of TMR Implementation

Given that the hardware is completed, the system is ready for software integration, which consists of two parts, the O/S and the HCI, which are detailed in Ref. 3. The VxWorks O/S previously constructed requires testing and installation onto the EPROMs. The TMR system would next be connected to a computer running the HCI. The implementation of the HCI will communicate to the TMR board via two serial cables. The handshaking between the HCI and hardware needs to be tested to ensure the communication paths are operating accurately and the data format being transmitted is correct. When these objectives are fulfilled, the system will be a fully functional TMR system that is able to detect and correct single errors in any of the processors and provide the data corresponding to that error to the HCI for analysis. The TMR system is then set for radiation testing and modifications for other uses.

2. Radiation Testing

The completion of the software and hardware integration will lead to the radiation testing of the design. This testing will investigate the survivability of the COTS devices utilized on the board and the ability of the design to handle SEUs. The test facility that will most likely be utilized for this is the cyclotron at the University of California at Davis. Additionally, the Naval Research Laboratory will allow utilization of a laser to implement SEU. The laser requires exact precision to impact registers on the devices tested.

APPENDIX A UPDATED TMR PLD FILES

Name   MemCont ;
PartNo   ATF22V10C-7PC ;
Created Date   5/27/00 ;
Revision 02 ;
Designer David Summers ;
Modified by Damen Hofheinz;
Company NPS ;
Assembly TMR R3081 ;
Location U54 ;
Device   g22v10 ;

```
/* ********************* INPUT PINS    ***********************/
PIN  1 = SYSCLK;   /* SYSCLK FR CPUA                         */
PIN  2 = INTCSN;   /* INTERRUPT CHIP SELECT                  */
PIN  3 = RAMCSN;   /* SRAM CHIP SELECT                       */
PIN  4 = TIMERCSN; /* TIMER CHIP SELECT                      */
PIN  5 = UARTCSN;  /* UART CHIP SELECT                       */
PIN  6 = EPROMCSN; /* EPROM CHIP SELECT                      */
PIN  7 = VOTBURSTN; /* VOTED BURST READ/WRITE NEAR           */
PIN  8 = VOTRDN;   /* VOTED READ                             */
PIN  9 = VOTWRN;   /* VOTED WRITE                            */
PIN 10 = CVOTERR;  /* CONTROL VOTER ERROR                    */
PIN 11 = DVOTERR;  /* DATA VOTER ERROR                       */
PIN 13 = RESETN;   /* SYNCHRONOUS SYSTEM RESET               */
PIN 19 = AVOTERR;  /* ADDRESS VOTER ERROR                    */

/* ********************* OUTPUT PINS    **********************/
PIN [14,15,16,17] = [COUNT3..0];  /* WAIT STATE GENERATOR    */
PIN 18 = CYCENDN;           /* CYCLE END SIGNAL              */
PIN 20 = BUSERRN;           /* BUS ERROR SIGNAL TO CPU       */
PIN 21 = ACKN;            /* ACK SIGNAL TO CPU               */
PIN 22 = RDCENN;            /* READ CLOCK ENABLE TO CPU       */
PIN 23 = VOTINTN;           /* VOTER INTERRUPT TO CPU        */

/***************** CONSTANT DEFINITIONS    *****************/
$DEFINE LOW  'B'0
$DEFINE HIGH 'B'1
```

```
/********************** LOGIC EQUATIONS  **********************/

/************************************************************/
/*                                                        */
/*        WAIT STATE COUNTER COUNT[3..0]                  */
/*                                                        */
/*THE PURPOSE OF THE WAIT STATE COUNTER IS TO PROVIDE     */
/*REFERENCE FOR THE MEMORY CONTROLLER SIGNALS.  IT        */
/*STARTS COUNTING  WHEN A READ OR WRITE CYCLE IS INITIATED */
/*AND RESETS WHEN THERE IS A RESET OR CYCLE ENDSIGNAL.    */
/*THE COUNTER USES THE SYSCLK AS ITS REFERENCE CLOCK.     */
/************************************************************/

COUNT0.D = RESETN & CYCENDN & (!VOTRDN # !VOTWRN) &
           (COUNT0 $ HIGH);
COUNT1.D = RESETN & CYCENDN & (!VOTRDN # !VOTWRN) &
           (COUNT1 $ COUNT0);
COUNT2.D = RESETN & CYCENDN & (!VOTRDN # !VOTWRN) &
           (COUNT2 $ (COUNT1 & COUNT0));
COUNT3.D = RESETN & CYCENDN & (!VOTRDN # !VOTWRN) &
           (COUNT3 $ (COUNT2 & COUNT1 & COUNT0));

COUNT0.OE=HIGH;
COUNT1.OE = HIGH;
COUNT2.OE = HIGH;
COUNT3.OE = HIGH;

COUNT0.AR = LOW;
COUNT1.AR = LOW;
COUNT2.AR = LOW;
COUNT3.AR = LOW;

COUNT0.SP = LOW;
COUNT1.SP = LOW;
COUNT2.SP = LOW;
COUNT3.SP = LOW;

FIELD CNTR = [COUNT3, COUNT2, COUNT1, COUNT0];
```

98

```
/* *****************************************************************/
/*                                                               */
/*              CYCLE END SIGNAL                                 */
/*                                                               */
/* THE PURPOSE OF THE CYCLE END SIGNAL IS TO SIGNAL THE END      */
/* OF THE  OF THE CURRENT BUS CYCLE IN ORDER TO RESET            */
/* THE COUNTER.  THIS RESETS ITSELF BY INCLUDING A REFERENCE*/
/*TO ITSELF IN THE EQUATIONS.                                   */
/* *****************************************************************/

CYCENDN.D = !(RESETN & CYCENDN & (
                (!RAMCSN & (CNTR : 'H'0) & !VOTRDN & VOTBURSTN)
              # (!RAMCSN & (CNTR : 'H'6) & !VOTRDN & !VOTBURSTN)
              # (!RAMCSN & (CNTR : 'H'0) & !VOTWRN)
              # (!EPROMCSN & (CNTR : 'H'1) & !VOTRDN & VOTBURSTN)
              # (!EPROMCSN & (CNTR : 'H'7) & !VOTRDN & !VOTBURSTN)
              # (!UARTCSN & (CNTR : 'H'0) & VOTBURSTN)
              # (!TIMERCSN & (CNTR : 'H'0) & VOTBURSTN)
              # (!INTCSN & (CNTR : 'H'0) & VOTBURSTN)
              # (CNTR : 'H'F)
              ));
CYCENDN.AR = LOW;
CYCENDN.SP = LOW;




/* *****************************************************************/
/*                                                               */
/*              READ CLOCK ENABLE                               */
/*                                                               */
/* THE READ CLOCK ENABLE SIGNAL IS USED BY THE CPU TO           */
/* STROBE DATA  OFF THE DATA BUS INTO ITS READ BUFFER.  THIS     */
/* SIGNAL IS STROBED ONE TIME FOR SINGLE READS AND FOUR         */
/* TIMES FOR QUAD READS.  ONLY RAM AND EPROM MEMORY            */
/*USE THE QUAD WORD READS.                                     */
/* *****************************************************************/

RDCENN.D = !(RESETN & CYCENDN & !VOTRDN & (

                (!RAMCSN & (
                        (CNTR : 'H'0)
                      # (!VOTBURSTN & (CNTR : 'H'2))
```

99

```
                    # (!VOTBURSTN & (CNTR : H'4))
                    # (!VOTBURSTN & (CNTR : H'6))
                    )
                )
            # (!EPROMCSN & (
                        (CNTR : H'1)
                    # (!VOTBURSTN & (CNTR : H'3))
                    # (!VOTBURSTN & (CNTR : H'5))
                    # (!VOTBURSTN & (CNTR : H'7))
                    )
            )
            # (!UARTCSN & (CNTR : H'0))
            # (!TIMERCSN & (CNTR : H'0))
            # (!INTCSN & (CNTR : H'0))
        ));
RDCENN.AR = LOW;
RDCENN.SP = LOW;




/* ************************************************************* */
/*                                                           */
/*                 ACKNOWLEDGE                               */
/*                                                           */
/* THE ACKNOWLEDGE SIGNAL IS USED BY THE MEMORY SYSTEM       */
/* TO LET THE CPU KNOW THAT IT HAS PROCESSED THE WRITE       */
/* CYCLE SUFFICIENTLY AND THE CPU MAY MOVE ON TO THE         */
/* NEXT CYCLE. THIS SIGNAL IS GENERATED IMPLICITLY ON        */
/* SINGLE DATUM READS AND NO SOONER THAN FOUR CLOCKS         */
/* BEFORE THE END OF THE LAST READ FOR BURSTS.               */
/* ************************************************************* */

ACKN.D = !(RESETN & CYCENDN & (
            (!RAMCSN & !VOTWRN & (CNTR : H'0))
        # (!RAMCSN & !VOTRDN & !VOTBURSTN & (CNTR : H'3))
        # (!EPROMCSN & !VOTRDN & !VOTBURSTN & (CNTR : H'4))
        # (!UARTCSN & !VOTWRN & VOTBURSTN & (CNTR : H'0))
        # (!TIMERCSN & !VOTWRN & (CNTR : H'0))
        # (!INTCSN & !VOTWRN & (CNTR : H'0))
            )
        );

ACKN.AR = LOW;
ACKN.SP = LOW;
```

```
/*****************************************************************/
/*              CHANGE MADE IN THIS SECTION              */
/*                    BUS ERROR                          */
/*                                                       */
/* THE BUS ERROR SIGNAL IS USED BY THE PROCESSOR TO END  */
/* A BUS CYCLE  THAT TRIES TO ACCESS AN ADDRESS THAT IS  */
/* NOT POPULATED IN THE  MEMORY SPACE.                   */
/*                                                       */
/* CHANGED COUNTER FROM H'F TO H'E'.                     */
/* *****************************************************************/

BUSERRN.D = !(RESETN & CYCENDN & (CNTR : 'H'E));
BUSERRN.AR = LOW;
BUSERRN.SP = LOW;



/* *****************************************************************/
/*                                                     */
/*              VOTER INTERRUPT SIGNAL                   */
/*                                                       */
/* THE VOTER INTERRUPT SIGNAL INFORMS THE CPU WHEN       */
/* A MISCOMPARE  HAPPENS IN ONE OF THE FPGAS.  THE SIGNAL IS  */
/* HELD UNTIL A INTCSN IS GENERATED BY THE ADDRESS       */
/*DECODER.                                               */
/* *****************************************************************/

VOTINTN.D = !((AVOTERR # CVOTERR # DVOTERR # !VOTINTN) & INTCSN);
VOTINTN.AR = LOW;
VOTINTN.SP = LOW;
```
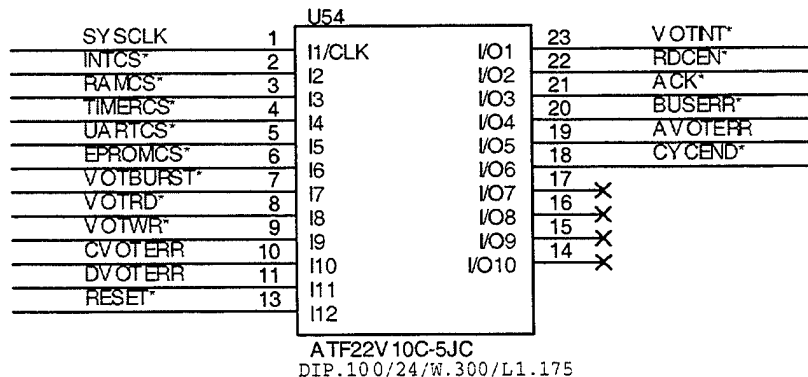

Figure A.1 MEMCONT PLD

101
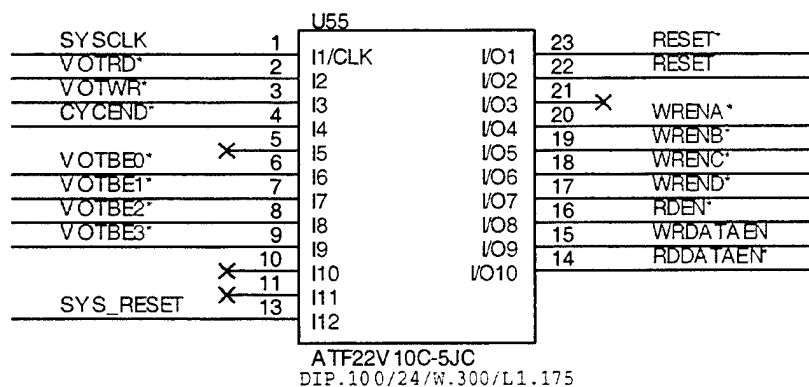
Figure A.2 MEMENABLE PLD

APPENDIX B TMR SYSTEM CONTROLLER FILES

This appendix contains the HDL code for the TMR system controller FPGA.

Table B.1. Lists the state machines in this appendix and the page they appear on.

Table B.1. TMR Files

```
// Author: Damen Hofheinz
//  UARTC.v
// created: 08/16/00 21:30:48
// INPUTS: CLK
// OUTPUTS: CUARTCSN, UARTEN, CUWREN, CTRLDATAU[7:0]
// CTRLADDR[2:0]
// INTERNAL SIGNALS:


    module Uartc (CLK, CTRLADDR, CTRLDATAU, CUARTCSN, CUWREN,
UARTEN);
        input  CLK;
        output [2:0]CTRLADDR;
        output [7:0]CTRLDATAU;
        output CUARTCSN;
        output CUWREN;
        output UARTEN;

        reg    [2:0]CTRLADDR, next_CTRLADDR;
        reg    [7:0]CTRLDATAU, next_CTRLDATAU;
        reg    CUARTCSN, next_CUARTCSN;
        reg    CUWREN, next_CUWREN;
        reg    UARTEN, next_UARTEN;


    // USER DEFINED ENCODED state machine: UARTC
    parameter S0 = 5'b00000,
            S1 = 5'b00001,
            S10 = 5'b01010,
            S11 = 5'b01011,
            S12 = 5'b01100,
            S13 = 5'b01101,
            S14 = 5'b01110,
            S15 = 5'b01111,
            S16 = 5'b10000,
            S17 = 5'b10001,
            S2 = 5'b00010,
            S2a = 5'b10010,
            s3 = 5'b00011,
            S4 = 5'b00100,
            S5 = 5'b00101,
            S6 = 5'b00110,
            S7 = 5'b00111,
            S8 = 5'b01000,
```

104

```verilog
        S9 = 5'b01001;
reg [4:0]CurrState_Sreg0, NextState_Sreg0;

// Diagram actions (continuous assignments allowed only: assign ...)
//diagram ACTIONS;

// Machine: Sreg0

// NextState logic (combinatorial)
always @ (CurrState_Sreg0)
begin
        case (CurrState_Sreg0)          // synopsys parallel_case full_case
                S0:
                begin
                        begin
                                NextState_Sreg0 = S1;
                        end
                end
                S1:
                begin
                        begin
                                NextState_Sreg0 = S2;
                                next_CUWREN = 1;
                                next_CUARTCSN = 0;
                                next_UARTEN = 0;
                        end
                end
                S10:
                begin
                        begin
                                NextState_Sreg0 = S11;
                                next_CUWREN = 0;
                        end
                end
                S11:
                begin
                        begin
                                NextState_Sreg0 = S12;
                                next_CUWREN = 1;
                        end
                end
                S12:
                begin
                        begin
```

```verilog
                NextState_Sreg0 = S13;
                next_CTRLADDR = 3'b001;
                next_CTRLDATAU = 8'b00000000;
        end
end
S13:
begin
        begin
                NextState_Sreg0 = S14;
                next_CUWREN = 0;
        end
end
S14:
begin
        begin
                NextState_Sreg0 = S15;
                next_CUWREN = 1;
        end
end
S15:
begin
        begin
                NextState_Sreg0 = S16;
                next_CUARTCSN = 1;
                next_UARTEN = 1;
        end
end
S16:
begin
        begin
                NextState_Sreg0 = S17;
        end
end
S17:
begin
        begin
                NextState_Sreg0 = S17;
        end
end
S2:
begin
        next_CTRLADDR = 3'b010;
        next_CTRLDATAU = 8'b10000000;
        begin
```

```
                        NextState_Sreg0 = S2a;
            end
end
S2a:
begin
        next_CUWREN = 0;
        begin
                NextState_Sreg0 = s3;
        end
end
s3:
begin
        begin
                NextState_Sreg0 = S4;
                next_CTRLADDR = 3'b011;
                next_CTRLDATAU = 8'b11000001;
                next_CUWREN = 1;
        end
end
S4:
begin
        begin
                NextState_Sreg0 = S5;
                next_CUWREN = 0;
        end
end
S5:
begin
        begin
                NextState_Sreg0 = S6;
                next_CUWREN = 1;
        end
end
S6:
begin
        begin
                NextState_Sreg0 = S7;
                next_CTRLADDR = 3'b100;
                next_CTRLDATAU = 8'b11000100;
        end
end
S7:
begin
        begin
```

```verilog
                                NextState_Sreg0 = S8;
                                next_CUWREN = 1;
                        end
                end
                S8:
                begin
                        begin
                                NextState_Sreg0 = S9;
                                next_CUWREN = 0;
                        end
                end
                S9:
                begin
                        begin
                                NextState_Sreg0 = S10;
                                next_CTRLADDR = 3'b000;
                                next_CTRLDATAU = 8'b01001000;
                        end
                end
        endcase
end

// Current State Logic (sequential)
always @ (posedge CLK)
begin
        CurrState_Sreg0 <= NextState_Sreg0;
end

// Registered outputs logic
always @ (posedge CLK)
begin
        CUWREN <= next_CUWREN;
        CUARTCSN <= next_CUARTCSN;
        UARTEN <= next_UARTEN;
        CTRLADDR <= next_CTRLADDR;
        CTRLDATAU <= next_CTRLDATAU;
end

endmodule
```

108

```verilog
// Author: Damen Hofheinz
// File: votmch.v
// created: 08/12/00 21:44:07
// INPUTS: CLK, CUARTINT, CPUDONE, INTCSN
// OUTPUTS: UARTINT, INTRCNTR[3:0]
// INTERNAL SIGNALS: NONE

        module   votmch   (CLK,   CPUDONE,   CUARTINT,   INTCSN,   INTRCNT,
UARTINT);
        input  CLK;
        input  CPUDONE;
        input  CUARTINT;
        input  INTCSN;
        output [3:0]INTRCNT;
        output  UARTINT;


        reg    [3:0]INTRCNT, next_INTRCNT;
        reg    UARTINT, next_UARTINT;



// USER DEFINED ENCODED state machine: Sreg0
parameter S1 = 2'b00,
        S2 = 2'b01,
        S3 = 2'b10,
        S4 = 2'b11;
reg [1:0]CurrState_Sreg0, NextState_Sreg0;

// Diagram actions (continuous assignments allowed only: assign ...)
//diagram ACTIONS;

// Machine: Sreg0

// NextState logic (combinatorial)
always @ (CPUDONE or CUARTINT or INTCSN or CurrState_Sreg0)
begin
        case (CurrState_Sreg0)          // synopsys parallel_case full_case
                S1:
                begin
                        if (!INTCSN)
                        begin
                                NextState_Sreg0 = S2;
                        end
                        else if (CUARTINT && !INTRCNT)
                        begin
```

```verilog
                        NextState_Sreg0 = S4;
                end
                else if (INTRCNT)
                begin
                        NextState_Sreg0 = S3;
                end
        end
        S2:
        begin
                if (INTCSN)
                begin
                        NextState_Sreg0 = S1;
                        next_INTRCNT = INTRCNT+1;
                end
        end
        S3:
        begin
                if (CPUDONE)
                begin
                        NextState_Sreg0 = S1;
                        next_INTRCNT = INTRCNT-1;
                end
        end
        S4:
        begin
                next_UARTINT = 1;
                if (!CUARTINT)
                begin
                        NextState_Sreg0 = S1;
                        next_UARTINT = 0;
                end
        end
    endcase
end

// Current State Logic (sequential)
always @ (posedge CLK)
begin
        CurrState_Sreg0 <= NextState_Sreg0;
end

// Registered outputs logic
always @ (posedge CLK)
begin
```

```
            INTRCNT <= next_INTRCNT;
            UARTINT <= next_UARTINT;
end

endmodule
```

```
module cntroller (CTRLADDR, CTRLDATAV, CUARTADSN, CUARTCSN,
CURDEN, FIFOCTRLA, FIFOCTRLB, FIFOCTRLC, FWRCLK1, HDREN, INTCNTR,
PROCXFER, SYSCLK, XFERCOMP);
        input   [3:0]INTCNTR;
        input   SYSCLK;
        output [2:0]CTRLADDR;
        output [7:0]CTRLDATAV;
        output CUARTADSN;
        output CUARTCSN;
        output CURDEN;
        output [9:0]FIFOCTRLA;
        output [9:0]FIFOCTRLB;
        output [9:0]FIFOCTRLC;
        output FWRCLK1;
        output HDREN;
        output PROCXFER;
        output XFERCOMP;

        reg    [2:0]CTRLADDR, next_CTRLADDR;
        reg    [7:0]CTRLDATAV, next_CTRLDATAV;
        reg    CUARTADSN, next_CUARTADSN;
        reg    CUARTCSN, next_CUARTCSN;
        reg    CURDEN, next_CURDEN;
        reg    [9:0]FIFOCTRLA, next_FIFOCTRLA;
        reg    [9:0]FIFOCTRLB, next_FIFOCTRLB;
        reg    [9:0]FIFOCTRLC, next_FIFOCTRLC;
        reg    FWRCLK1, next_FWRCLK1;
        reg    HDREN, next_HDREN;
        reg    PROCXFER, next_PROCXFER;
        reg    XFERCOMP, next_XFERCOMP;

// diagram signal declarations
```

112

```verilog
reg [7:0]BYTCNT, next_BYTCNT;
reg [1:0]CPUCNT, next_CPUCNT;
reg [1:0]CPUHDR, next_CPUHDR;
reg FIFOCOMP, next_FIFOCOMP;
reg FIFOENGA, next_FIFOENGA;
reg FIFOENGB, next_FIFOENGB;
reg FIFOENGC, next_FIFOENGC;
reg [7:0]FIFOHDR, next_FIFOHDR;
reg HDONE, next_HDONE;

// USER DEFINED ENCODED state machine: Sreg0
parameter XFERP = 1'b1,
        XFERSTRT = 1'b0;
reg CurrState_Sreg0, NextState_Sreg0;

// USER DEFINED ENCODED state machine: Sreg1
parameter CPUA = 3'b001,
        CPUB = 3'b010,
        CPUC = 3'b011,
        DONE = 3'b100,
        STRT = 3'b000,
        WAITST = 3'b101;
reg [2:0]CurrState_Sreg1, NextState_Sreg1;

// USER DEFINED ENCODED state machine: Sreg2
parameter FIFOST0 = 3'b001,
        FIFOST1 = 3'b010,
        FIFOST2 = 3'b011,
        FIFOST3 = 3'b100,
        FIFOST4 = 3'b101,
        FIFOSTART = 3'b000;
reg [2:0]CurrState_Sreg2, NextState_Sreg2;

// USER DEFINED ENCODED state machine: Sreg3
parameter BTYE5 = 4'b0110,
        BYTE10 = 4'b1011,
        BYTE2 = 4'b0001,
        BYTE3 = 4'b0010,
        BYTE4 = 4'b0100,
        BYTE6 = 4'b0111,
        BYTE7 = 4'b1000,
        BYTE8 = 4'b1001,
        BYTE9 = 4'b1010,
        BYTSTRT = 4'b0000,
```

```
        S1 = 4'b1100;
reg [3:0]CurrState_Sreg3, NextState_Sreg3;


// Machine: VOTE
// NextState logic (combinatorial)
always @ (INTCNTR or CurrState_Sreg0)
begin
        case (CurrState_Sreg0)          // synopsys parallel_case full_case
                XFERP:
                begin
                        if (XFERCOMP)
                        begin
                                NextState_Sreg0 = XFERSTRT;
                        end
                end
                XFERSTRT:
                begin
                        next_PROCXFER = 0;
                        if (INTCNTR)
                        begin
                                NextState_Sreg0 = XFERP;
                                next_PROCXFER = 1;
                        end
                        else if (!INTCNTR)
                        begin
                                NextState_Sreg0 = XFERSTRT;
                        end
                end
        endcase
end


// Current State Logic (sequential)
always @ (posedge SYSCLK)
begin
        CurrState_Sreg0 <= NextState_Sreg0;
end


// Registered outputs logic
always @ (posedge SYSCLK)
begin
        PROCXFER <= next_PROCXFER;
end
```

```
// Machine: CPU
// NextState logic (combinatorial)
always @ (INTCNTR or CurrState_Sreg1)
begin
        case (CurrState_Sreg1)           // synopsys parallel_case full_case
                CPUA:
                begin
                        if (CPUCNT==2'b01)
                         · begin
                                        NextState_Sreg1 = CPUB;
                                        next_CPUHDR[1:0] = 2'b01;
                                        next_FIFOENGB = 1;
                                        next_FIFOENGA = 0;
                                end
                end
                CPUB:
                begin
                        if (CPUCNT==2'b10)
                        begin
                                        NextState_Sreg1 = CPUC;
                                        next_CPUHDR[1:0] = 2'b10;
                                        next_FIFOENGC = 1;
                                        next_FIFOENGB = 0;
                                end
                end
                CPUC:
                begin
                        if (CPUCNT==2'b11)
                        begin
                                        NextState_Sreg1 = DONE;
                                        next_FIFOENGC = 0;
                                        next_XFERCOMP = 1;
                                end
                end
                DONE:
                begin
                        begin
                                        NextState_Sreg1 = WAITST;
                                end
                end
                STRT:
                begin
                        next_CPUHDR[1:0] = 2'b11;
                        if (PROCXFER==1)
```

```
                begin
                        NextState_Sreg1 = CPUA;
                        next_FIFOENGA = 1;
                        next_CPUHDR[1:0] = 2'b00;
                end
        end
        WAITST:
        begin
                begin
                        NextState_Sreg1 = STRT;
                end
        end
        endcase
end




// Current State Logic (sequential)
always @ (posedge SYSCLK)
begin
        CurrState_Sreg1 <= NextState_Sreg1;
end

// Registered outputs logic
always @ (posedge SYSCLK)
begin
        XFERCOMP <= next_XFERCOMP;
        CPUHDR <= next_CPUHDR;
        FIFOENGB <= next_FIFOENGB;
        FIFOENGA <= next_FIFOENGA;
        FIFOENGC <= next_FIFOENGC;
end




// Machine: FIFOXFER
// NextState logic (combinatorial)
always @ (INTCNTR or CurrState_Sreg2)
begin
        case (CurrState_Sreg2)           // synopsys parallel_case full_case
                FIFOST0:
                begin
                        if (FIFOENGC==1 && HDONE==1 && FIFOCOMP)
                        begin
                                NextState_Sreg2 = FIFOST1;
```

116

```verilog
                        next_FIFOCTRLC[9:0] = 10'b1100111111;
                        next_FIFOHDR = 8'b00000100;
                end
                else if (FIFOENGB==1 && HDONE==1 && FIFOCOMP)
                begin
                        NextState_Sreg2 = FIFOST1;
                        next_FIFOCTRLB[9:0] = 10'b1100111111;
                        next_FIFOHDR = 8'b00000100;
                end
                else if (FIFOENGA==1 && HDONE==1 && FIFOCOMP)
                begin
                        NextState_Sreg2 = FIFOST1;
                        next_FIFOCTRLA[9:0] = 10'b1100111111;
                        next_FIFOHDR = 8'b00000100;
                end
                else if (FIFOCOMP!=1)
                begin
                        NextState_Sreg2 = FIFOST0;
                        next_FIFOHDR = 8'b11000000;
                end
        end
end
FIFOST1:
begin
        if (FIFOENGA==1 && HDONE && FIFOCOMP)
        begin
                NextState_Sreg2 = FIFOST2;
                next_FIFOCTRLA[9:0] = 10'b1111001111;
                next_FIFOHDR = 8'b00001000;
        end
        else if (FIFOENGB==1 && HDONE && FIFOCOMP)
        begin
                NextState_Sreg2 = FIFOST2;
                next_FIFOCTRLB[9:0] = 10'b1111001111;
                next_FIFOHDR = 8'b00001000;
        end
        else if (FIFOENGC==1 && HDONE && FIFOCOMP)
        begin
                NextState_Sreg2 = FIFOST2;
                next_FIFOCTRLC[9:0] = 10'b1111001111;
                next_FIFOHDR = 8'b00001000;
        end
        else if (FIFOCOMP!=1)
        begin
                NextState_Sreg2 = FIFOST1;
```

117

```
                        next_FIFOHDR = 8'b00000100;
                end
        end
FIFOST2:
begin
        if (FIFOENGA==1 && HDONE && FIFOCOMP)
        begin
                NextState_Sreg2 = FIFOST3;
                next_FIFOCTRLA[9:0] = 10'b1111110011;
                next_FIFOHDR = 8'b00001100;
        end
        else if (FIFOENGB==1 && HDONE && FIFOCOMP)
        begin
                NextState_Sreg2 = FIFOST3;
                next_FIFOCTRLB[9:0] = 10'b1111110011;
                next_FIFOHDR = 8'b00001100;
        end
        else if (FIFOENGC==1 && HDONE && FIFOCOMP)
        begin
                NextState_Sreg2 = FIFOST3;
                next_FIFOCTRLC[9:0] = 10'b1111110011;
                next_FIFOHDR = 8'b00001100;
        end
        else if (FIFOCOMP!=1)
        begin
                NextState_Sreg2 = FIFOST2;
                next_FIFOHDR = 8'b00001000;
        end
end
FIFOST3:
begin
        if (FIFOENGC==1 && HDONE && FIFOCOMP)
        begin
                NextState_Sreg2 = FIFOST4;
                next_FIFOCTRLC[9:0] = 10'b1111111100;
                next_CPUCNT = CPUCNT+1;
        end
        else if (FIFOENGA==1 && HDONE & FIFOCOMP)
        begin
                NextState_Sreg2 = FIFOST4;
                next_FIFOCTRLA[9:0] = 10'b1111111100;
                next_CPUCNT = CPUCNT+1;
        end
        else if (FIFOENGB==1 && HDONE && FIFOCOMP)
```

```verilog
            begin
                    NextState_Sreg2 = FIFOST4;
                    next_FIFOCTRLB[9:0] = 10'b1111111100;
                    next_CPUCNT = CPUCNT+1;
            end
            else if (FIFOCOMP!=1)
            begin
                    NextState_Sreg2 = FIFOST3;
                    next_FIFOHDR = 8'b00001100;
            end
    end
FIFOST4:
begin
            if (CPUCNT==2'b11 && FIFOCOMP==1)
            begin
                    NextState_Sreg2 = FIFOSTART;
            end
            else if (CPUCNT!=2'b11 && FIFOCOMP==1)
            begin
                    NextState_Sreg2 = FIFOSTART;
            end
    end
FIFOSTART:
begin
            if (FIFOENGC==1 &&  HDONE)
            begin
                    NextState_Sreg2 = FIFOST0;
                    next_FIFOCTRLC[9:0] = 10'b0011111111;
                    next_FIFOHDR = 8'b11000000;
            end
            else if (FIFOENGA==1 && HDONE)
            begin
                    NextState_Sreg2 = FIFOST0;
                    next_FIFOCTRLA[9:0] = 10'b0011111111;
                    next_FIFOHDR = 8'b11000000;
            end
            else if (FIFOENGB==1 && HDONE)
            begin
                    NextState_Sreg2 = FIFOST0;
                    next_FIFOCTRLB[9:0] = 10'b0011111111;
                    next_FIFOHDR = 8'b11000000;
            end
    end
endcase
```

```verilog
end

// Current State Logic (sequential)
always @ (posedge SYSCLK)
begin
        CurrState_Sreg2 <= NextState_Sreg2;
end

// Registered outputs logic
always @ (posedge SYSCLK)
begin
        FIFOCTRLC <= next_FIFOCTRLC;
        FIFOCTRLB <= next_FIFOCTRLB;
        FIFOCTRLA <= next_FIFOCTRLA;
        FIFOHDR <= next_FIFOHDR;
        CPUCNT <= next_CPUCNT;
end

// Machine: BYTE
// NextState logic (combinatorial)
always @ (INTCNTR or CurrState_Sreg3)
begin
        case (CurrState_Sreg3)          // synopsys parallel_case full_case
                BTYE5:
                begin
                        begin
                                NextState_Sreg3 = BYTE6;
                                next_FWRCLK1 = 0;
                                next_HDREN = 0;
                                next_HDONE = 1;
                        end
                end
                BYTE10:
                begin
                        begin
                                NextState_Sreg3 = BYTE7;
                                next_BYTCNT = BYTCNT+1;
                                next_FWRCLK1 = 0;
                        end
                end
                BYTE2:
                begin
                        next_CUARTCSN = 1;
                        next_CTRLADDR[2:0] = 3'b000;
```

120

```
                begin
                        NextState_Sreg3 = BYTE3;
                end
        end
        BYTE3:
        begin
                next_CUARTADSN = 1;
                begin
                        NextState_Sreg3 = BYTE4;
                        next_CTRLDATAV = CPUHDRlFIFOHDR;
                        next_HDREN = 1;
                        next_FWRCLK1 = 0;
                end
        end
        BYTE4:
        begin
                next_CURDEN = 1;
                next_FWRCLK1 = 1;
                begin
                        NextState_Sreg3 = BTYE5;
                        next_BYTCNT = 1;
                end
        end
        BYTE6:
        begin
                begin
                        NextState_Sreg3 = BYTE7;
                        next_FIFOCOMP = 0;
                end
        end
        BYTE7:
        begin
                next_FWRCLK1 = 1;
                next_HDONE = 0;
                begin
                        NextState_Sreg3 = BYTE8;
                end
        end
        BYTE8:
        begin
                begin
                        NextState_Sreg3 = BYTE9;
                        next_BYTCNT = BYTCNT+1;
                        next_FWRCLK1 = 0;
```

```
                        end
                end
                BYTE9:
                begin
                        if (BYTCNT==8'b01010010)
                        begin
                                NextState_Sreg3 = S1;
                                next_FIFOCOMP = 1;
                                next_FWRCLK1 = 1;
                        end
                        else if (BYTCNT<8'b01010010)
                        begin
                                NextState_Sreg3 = BYTE10;
                                next_FWRCLK1 = 1;
                        end
                end
                BYTSTRT:
                begin
                        next_CUARTCSN = 0;
                        next_CUARTADSN = 0;
                        next_CURDEN = 0;
                        if (FIFOENGA||FIFOENGB||FIFOENGC)
                        begin
                                NextState_Sreg3 = BYTE2;
                                next_BYTCNT = 0;
                        end
                end
                S1:
                begin
                        begin
                                NextState_Sreg3 = BYTSTRT;
                        end
                end
        endcase
end

// Current State Logic (sequential)
always @ (posedge SYSCLK)
begin
        CurrState_Sreg3 <= NextState_Sreg3;
end

// Registered outputs logic
always @ (posedge SYSCLK)
```

```
begin
        FWRCLK1 <= next_FWRCLK1;
        HDREN <= next_HDREN;
        CUARTCSN <= next_CUARTCSN;
        CTRLADDR <= next_CTRLADDR;
        CUARTADSN <= next_CUARTADSN;
        CTRLDATAV <= next_CTRLDATAV;
        CURDEN <= next_CURDEN;
        HDONE <= next_HDONE;
        BYTCNT <= next_BYTCNT;
        FIFOCOMP <= next_FIFOCOMP;
end
```

```verilog
//  Author: Damen Hofheinz
//  File: Collct.v
// created: 08/12/00 21:37:16
// INPUTS: CLK, VOTINT
// OUTPUTS: FIFOCTRLA, FIFOCTRLB, FIFOCTRLC
// INTERNAL SIGNALS: COLCNT[7:0]
//
module Collct (CLK, FIFOCTRLA, FIFOCTRLB, FIFOCTRLC, VOTINT);
input   CLK;
input   VOTINT;
output FIFOCTRLA;
output FIFOCTRLB;
output FIFOCTRLC;

reg    FIFOCTRLA;
reg    FIFOCTRLB;
reg    FIFOCTRLC;

// diagram signal declarations
reg  [7:0]COLCNT, next_COLCNT;

// USER DEFINED ENCODED state machine: COLLECT
parameter S1 = 2'b00,
        S2 = 2'b01,
        S3 = 2'b10,
        S4 = 2'b11;
reg [1:0]CurrState_Sreg0, NextState_Sreg0;

// Diagram actions (continuous assignments allowed only: assign ...)
//diagram ACTIONS;

// Machine: Sreg0

// NextState logic (combinatorial)
always @ (VOTINT or CurrState_Sreg0)
begin
        case (CurrState_Sreg0)          // synopsys parallel_case full_case
                S1:
                begin
                        if (!VOTINT)
                        begin
                                NextState_Sreg0 = S2;
                        end
                end
```
124

```verilog
            S2:
            begin
                    begin
                            NextState_Sreg0 = S3;
                            FIFOCTRLA=1;
                            FIFOCTRLB=1;
                            FIFOCTRLC=1;
                            next_COLCNT = 0;
                    end

            end
            S3:
            begin
                    next_COLCNT = COLCNT+1;
                    if (COLCNT==8'b10100100)
                    begin
                            NextState_Sreg0 = S4;
                    end

            end
            S4:
            begin
                    begin
                            NextState_Sreg0 = S1;
                            FIFOCTRLA=0;
                            FIFOCTRLB=0;
                            FIFOCTRLC=0;
                    end

                    end
            endcase
    end

// Current State Logic (sequential)
always @ (posedge CLK)
begin
        CurrState_Sreg0 <= NextState_Sreg0;
end

// Registered outputs logic
always @ (posedge CLK)
begin
        COLCNT <= next_COLCNT;
end

endmodule
```

```
// Author: Damen Hofheinz
// File: MODECNTRL.v
// created: 08/15/00 17:33:14
// INPUTS: UARTINT, CLK, AIN[7:0]
// OUTPUTS: CUARTADSN, FORCE, INO, CTRLADDR[2:0], CURDENN,
//CUARTCSN, MDCTRL[7:0]
// INTERNAL SIGNALS: NONE


module rdmach (AIN, BRDRST, CLK, CTRLADDR, CUARTADSN,
CUARTCSN, CURDENN, FORCE, INO, MDCTRL, SYSRST, UARTINT);
        input   [7:0]AIN;
        input   CLK;
        input   UARTINT;
        output  BRDRST;
        output  [2:0]CTRLADDR;
        output  CUARTADSN;
        output  CUARTCSN;
        output  CURDENN;
        output  FORCE;
        output  INO;
        output  [7:0]MDCTRL;
        output  SYSRST;

        reg     BRDRST;
        reg     [2:0]CTRLADDR;
        reg     CUARTADSN, next_CUARTADSN;
        reg     CUARTCSN;
        reg     CURDENN;
        reg     FORCE;
        reg     INO;
        reg     [7:0]MDCTRL, next_MDCTRL;
        reg     SYSRST;


        // SYMBOLIC ENCODED state machine: Sreg0
        parameter S1 = 4'b0000,
                S10 = 4'b0001,
                S2 = 4'b0010,
                S3 = 4'b0011,
                S4 = 4'b0100,
                S5 = 4'b0101,
                S6 = 4'b0110,
                S7 = 4'b0111,
```

126

```verilog
        S8 = 4'b1000,
        S9 = 4'b1001;
reg [3:0]CurrState_Sreg0, NextState_Sreg0;



// Machine: MODECNTRL
// NextState logic (combinatorial)
always @ (AIN or UARTINT or CurrState_Sreg0)
begin
        case (CurrState_Sreg0)          // synopsys parallel_case full_case
                S1:
                begin
                        if (UARTINT)
                        begin
                                NextState_Sreg0 = S2;
                                CUARTCSN=1;
                                INO=1;
                                FORCE=0;
                                SYSRST=0;
                                BRDRST=0;
                        end
                end
                S10:
                begin
                        FORCE=1;
                end
                S2:
                begin
                        begin
                                NextState_Sreg0 = S3;
                                CTRLADDR[2:0]=3'B010;
                        end
                end
                S3:
                begin
                        begin
                                NextState_Sreg0 = S4;
                                next_CUARTADSN = 1;
                        end
                end
                S4:
                begin
                        begin
                                NextState_Sreg0 = S5;
```

127

```verilog
                              CURDENN=1;
                      end
              end
              S5:
              begin
                      begin
                              NextState_Sreg0 = S6;
                              next_MDCTRL[7:0] = AIN;
                      end
              end
              S6:
              begin
                      begin
                              NextState_Sreg0 = S7;
                              CURDENN=0;
                              CUARTCSN=0;
                              INO=0;
                      end
              end
              S7:
              begin
                      if (MDCTRL[0])
                      begin
                              NextState_Sreg0 = S10;
                      end
                      else if (MDCTRL[2])
                      begin
                              NextState_Sreg0 = S8;
                      end
                      else if (MDCTRL[1])
                      begin
                              NextState_Sreg0 = S9;
                      end
              end
              S8:
              begin
                      BRDRST=1;
              end
              S9:
              begin
                      SYSRST=1;
              end
      endcase
end
```

```verilog
// Current State Logic (sequential)
always @ (posedge CLK)
begin
        CurrState_Sreg0 <= NextState_Sreg0;
end

// Registered outputs logic
always @ (posedge CLK)
begin
        CUARTADSN <= next_CUARTADSN;
        MDCTRL <= next_MDCTRL;
end

endmodule
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C PROGRAM FILES

This Appendix lists the Makefile and program files utilized with the Generic

Cross Compiler provided by IDT.

```
/***************************************************************/
/*Name:      makefile                                         */
/*Written by: Damen Hofheinz                                  */
/*Date:      9/28/00                                          */
/*Name:      Makefile                                         */
/*Function: This file is utilized by the GCC compiler to assemble and link the  */
/*          program file.                                     */
/*                                                            */
/*                                                            */
/***************************************************************/


            /* NAME OF FILE TO ASSEMBLE              */
            /* DEFINES DIRECTORY OF FILE LOCATIONS        */

EXE=rom
STARTUPDIR=/IDTC5.1
INCDIR=/IDTC5.1
LIBDIR=/IDTC5.1
OPTFLAGS=-O2

            /* DEFINES NAME OF OBJECT FILE              */
OBJS= $(EXE).o

            /* DEFINES NAME OF MOTOROLA S-RECORD FILE  TO MAKE*/
all: $(EXE).sre


            /* CONSTRUCTS .SRE FILE                      */
$(EXE).sre:   $(EXE)
       objcopy -O srec $(EXE) $(EXE).sre


            /* CREATES .EXE FILE FROM OBJECT FILE  AND LINKS */
            /* TO ADDR 0XBFC00000 */
$(EXE):  $(OBJS)
```

131

```
gcc -Wl,-M -nostdlib $(OPTFLAGS) -L$(LIBDIR) -o $(EXE) -Ttext 0xbfc00000
$(OBJS) -lkil -lc -lm -llnk -lgcc > $(EXE).map


                /* CREATES OBJECT FILE FROM ASSEMBLY LANGUAGE FILE */
$(EXE).o: $(EXE).S
gcc -nostdinc $(OPTFLAGS) -I$(INCDIR) -c -g -O -Wa,-alh $(EXE).S > $(EXE).lst


                /* OPTION TO DELETE FILES FROM LAST RUN */
clean:
        del *.o
        del $(EXE).sre
        del $(EXE)
```

```
/****************************************************************/
/*Name:        rom.s                                         */
/*Written by:  Damen Hofheinz                                */
/*Date:        9/28/00                                       */
/*                                            */
/*Function:  This program initializes the IDT3081 processor and reads program  */
/*           instructions from ROM.  The program then writes data to an         */
/*           address location in the KSEG0 memory segment that is              */
/*           incremented after each write.                                     */
/*                                                                             */
/****************************************************************/
/* startup code, noncacheable, no exceptions
       */
  #include "mips.h"
  #include "machine.h"
  #include "iregdef.h"
  #include "idtcpu.h"
  #include "idtmon.h"


        .text              /*  defines text section                   */
reset_exception:           /*  aligns instruction to reset address    */
        j       start      /* Upon reset program jumps to start       */
        .align  8
        .set noreorder
        nop ; .set reorder
        .align  7
general_exception:         /* If the processor has a general          */
        j       _exit      /* exception then it performs a jump to    */
                           /* the _exit label                         */


        .globl  start      /* global declaration of the label start   */
        .ent    start

start:
        .set    noreorder
                        /* LOAD SETUP DATA INTO REGISTER VO        */
                        /* SETUP DATA IS FORMED BY ORing   DEFINDED */
                        /* VARIABLE IN MIPS.H.  SETUP IS: CP0 USABLE AND IN */
                        /* KERNEL MODE, PROCESSOR IN BOOTSTRAP     */
                        /* MODE AND CLEAR INTR MASKS.              */


                li      v0,SR_PE|SR_CU0|SR_BEV
```

```
                    /* LOAD STATUS REGISTER WITH SETUP DATA    */
        mtc0    v0,C0_SR
        nop


                    /* CLEAR SOFTWARE INTERRUPTS               */
                    /* REQUIRED NOPS DUE TO REGISTER           */
                    /* WRITE INSTRUCTION                       */
        mtc0    zero,C0_CAUSE
        nop
        nop
        .set reorder

                    /* LOAD BEGINNING ADDR IN T0               */
        li      t0,0xa0001000


                    /* ROM WRITE LOOP                          */
                    /* STORE DATA TO ADDRESS POINTED TO BY     */
                    /* REGISTER T0.  ADD 4 TO ADDRESS STORED IN   */
                    /* REGISTER T0, THEN BRANCH TO LP.         */

lp:
        sw   zero,(t0)
        addu t0,4
         and t0,~0x2000
        b       lp

                    /* CONTINUOS LOOP IF PROGRAM EXPERIENCES A */
                    /* GENERAL EXCEPTION                     */
_exit:
        b       _exit
.end start
```

```
/******************************************************/
/*Name:      ramwrite.s                                */
/*Written by: Damen Hofheinz                           */
/*Date:       9/28/00                                  */
/*                                                     */
/*Function: This program initializes the IDT3081 processor, reads program */
/*          instructions from ROM, and writes to RAM.  The program        */
/*          writes data (0xdeadbeef) to a RAM address location in the KSEG1 */
/*          memory segment that is incremented.        */
/*                                                     */
/******************************************************/
/* startup code, noncacheable, no exceptions */
  #include "mips.h"
  #include "machine.h"
  #include "iregdef.h"
  #include "idtcpu.h"
  #include "idtmon.h"


        .text
reset_exception:
        j       start
        .align  8
        .set noreorder
        nop ; .set reorder
        .align  7
general_exception:
        j       _exit

        .globl  start
        .ent    start


start:
        .set    noreorder
                        /* LOAD SETUP DATA INTO REGISTER VO         */
                        /* SETUP DATA IS FORMED BY ORing   DEFINDED */
                        /* VARIABLE IN MIPS.H.  SETUP IS: CP0 USABLE AND IN */
                        /* KERNEL MODE, PROCESSOR IN BOOTSTRAP      */
                        /* MODE AND CLEAR INTR MASKS.               */


                li      v0,SR_PE|SR_CU0|SR_BEV
```

```
                    /* LOAD STATUS REGISTER WITH SETUP DATA     */
mtc0    v0,C0_SR
nop


                    /* CLEAR SOFTWARE INTERRUPTS                */
                    /* REQUIRED NOPS DUE TO REGISTER            */
                    /* WRITE INSTRUCTION                        */
mtc0    zero,C0_CAUSE
nop
nop
.set reorder


                    /* LOAD BEGINNING RAM ADDRESS               */
                    /* LOAD DATA IN TEMPORARY REGISTER          */
li      t0,0x80000000
li      t1,0xdeadbeef


                    /* RAM WRITE LOOP                           */
                    /* STORE DATA TO ADDRESS POINTED TO BY      */
                    /* REGISTER T0.  ADD 4 TO ADDRESS STORED IN */
                    /* REGISTER T0, THEN BRANCH TO LP.          */
lp:
    sw  t1,(t0)
    addu t0,4
    and t0,~0x6000
    b       lp

_exit:
    b       _exit
    .end start
```

```
/**************************************************************/
/*Name:       uart.s                                          /
/*Written by: Damen Hofheinz                                 */
/*Date:         10/29/00                                     */
/*                                                            */
/*Function:  This program initializes the IDT3081 processor, reads program  */
/*               instructions from ROM which initializes the UART.  The program  */
/*               setups the UART in polling mode, 8 bits, no parity, 9600 baud.  */
/*               A continuous loop writes data to the UART output address  */
/*               that outputs ACSII character 0x23 or '#'.    */
/*                                                            */
/**************************************************************/

    #include "mips.h"
    #include "machine.h"
    #include "iregdef.h"
    #include "idtcpu.h"
    #include "idtmon.h"


            .text
reset_exception:
            j       _exit
            .align  8
            .set noreorder
            nop ; .set reorder
            .align  7
general_exception:
            .set noreorder
            nop
            li      t0,0xfeedface
            nop
            .set reorder
            j       lp

            .globl  start
            .ent    start

start:
            .set    noreorder
                        /* LOAD SETUP DATA INTO REGISTER V0      */
                        /* SETUP DATA IS FORMED BY ORing   DEFINDED  */
                        /* VARIABLE IN MIPS.H.  SETUP IS: CP0 USABLE AND IN */
                        /* KERNEL MODE, PROCESSOR IN BOOTSTRAP      */
                        /* MODE AND CLEAR INTR MASKS.               */
```

137

```
        li      v0,SR_PE|SR_CU0|SR_BEV


                /* LOAD STATUS REGISTER WITH SETUP DATA     */
mtc0   v0,C0_SR
nop


                /* CLEAR SOFTWARE INTERRUPTS     */
                /* REQUIRED NOPS DUE TO REGISTER */
                /* WRITE INSTRUCTION             */
mtc0   zero,C0_CAUSE
nop
nop
.set reorder


                /*FIFO CONTROL REGISTER*/
                /* ENABLE FIFOs  */

li      t1,0xc1c1c1c1
nop
sw      t1,0xBFE00008
nop
                /* LINE CONTROL REGISTER */
                /* SET DLAB TO ONE AND DEFINE WORD LENGTH AS */
                /* 8 BITS */
li      t1,0x83838383
nop
sw      t1,0xBFE0000c
nop
                /* MODEM CONTROL REGISTER */
                /* SET UART IN POLLING MODE */
li      t1,0x00000000
nop
sw      t1,0xBFE00010
nop
                /* DIVISOR LATCH LSB REGISTER      */
                /* SET LOWER DIVISOR TO 72 DECIMAL */
li      t1,0x48484848
nop
sw      t1,0xBFE00000
```

```
        nop
                        /* DIVISOR LATCH USB REGISTER       */
              .         /* CLEAR DIVISOR REGISTER           */
        li      t1,0x00000000
        nop
        sw      t1,0xBFE00004
        nop


                        /* LINE CONTROL REGISTER            */
                        /* SET DLAB TO ZERO TO ACCESS       */
                        /*RCV/TRANSMIT REGISTERS            */
        li      t1,0x03030303
        nop
        sw      t1,0xBFE0000C
        nop
                        /* INTERRUPT ENABLE REGISTER        */
                        /* DISABLE INTERRUPTS               */
        li      t1,0x00000000
        nop
        sw      t1,0xBFE00004
        nop


                        /* WRITE LOOP. LOAD DATA INTO REGISTER      */
                        /* T1. STORE DATA (WRITE) TO UART TRANSMIT   */
                        /* REGISTER.                                */

lp:
        li      t1,0x23222223
        nop
        sw      t1,0xBFE00000
        nop

        nop
        b       lp              /* branch back to lp */
        nop


_exit:
        b       _exit
        .end start
/****************************************************************** */
/*Name:      uartio.s                                             */
/*Written by:  Damen Hofheinz                                     */
/*Date:        11/05/00                                           */
```

```
/*                                                            */
/*Function:  This program initializes the IDT3081 processor, reads program    */
/*           instructions from ROM which initializes the UART.  The program   */
/*           setups the UART in autoflow mode, 8 bits, no parity, 9600 baud.  */
/*           A loop reads in data from the UART, adds 30 to this data and      */
/*           ouputs the modified data to the UART.*/
/*                                                            */
/*********************************************************************/


        /* startup code, noncacheable, no exceptions */
        #include "mips.h"
        #include "machine.h"
        #include "iregdef.h"
        #include "idtcpu.h"
        #include "idtmon.h"


                .text
reset_exception:
                j       start
                .align  8
                .set noreorder
                nop ; .set reorder
                .align  7
general_exception:
                .set noreorder
                nop
                li      t0,0xfeedface
                mfc0    v0,C0_CAUSE
                nop
                .set reorder
                j       lp

                .globl  start
                .ent    start

start:
                .set    noreorder
                        /* LOAD SETUP DATA INTO REGISTER VO        */
                        /* SETUP DATA IS FORMED BY ORing   DEFINDED    */
                        /* VARIABLE IN MIPS.H.  SETUP IS: CP0 USABLE AND IN */
                        /* KERNEL MODE, PROCESSOR IN BOOTSTRAP     */
                        /* MODE AND CLEAR INTR MASKS.              */
                li      v0,SR_PE|SR_CU0|SR_CU1|SR_BEV|SR_IMASK8
```

```
                /* LOAD STATUS REGISTER WITH SETUP DATA       */
        mtc0    v0,C0_SR
        nop
        mtc0    zero,C0_CAUSE               /* clear software interrupts */
        nop                             /* rqd nops */
        nop
        .set reorder

        /*FIFO CONTROL REGISTER*/
        /* ENABLE FIFOs */
        li      t1,0xc7c7c7c7
        nop
        sw      t1,0xBFE00008
        nop


                /* LINE CONTROL REGISTER */
                /* SET DLAB TO ONE AND DEFINE WORD LENGTH AS */
                /* 8 BITS */

        li      t1,0x83838383
        nop
        sw      t1,0xBFE0000c
        nop
                /* MODEM CONTROL REGISTER */
                /* SET UART IN POLLING MODE */

        li      t1,0x23232323
        nop
        sw      t1,0xBFE00010
        nop

        /* DIVISOR LATCH LSB REGISTER         */
        /* SET LOWER DIVISOR TO 72 DECIMAL */
li      t1,0x48484848
nop
sw      t1,0xBFE00000
nop
                /* DIVISOR LATCH USB REGISTER        */
                /* CLEAR DIVISOR REGISTER            */
li      t1,0x00000000
nop
sw      t1,0xBFE00004
```

```
nop

                    /* LINE CONTROL REGISTER        */
                    /* SET DLAB TO ZERO TO ACCESS   */
                    /*RCV/TRANSMIT REGISTERS         */
li      t1,0x03030303
nop
sw      t1,0xBFE0000C
nop
                    /* INTERRUPT ENABLE REGISTER    */
                    /* DISABLE INTERRUPTS           */
li      t1,0x00000000
nop
sw      t1,0xBFE00004
nop

/* OUTPUTS START CHARACTER # TO SCREEN  */
lp:
        li      t1,0x23222223
        nop
        li      t4,0x00000060
        nop
        li      t5,0x00000001
        nop




/*  LOOP TO READ IN DATA, ADD 30, AND OUTPUT DATA  */
xt:     lw      t2,0xBFE00014
        nop
        andi    t2,t2,0x00000060
        nop
        bne     t2,t4,xt
        nop
        sw      t1,0xBFE00000
        nop

gt:     lw      t3,0xBFE00014
        nop
        andi    t3,t3,0x00000001
        nop
        bne     t3,t5,gt
        nop
```

```
        lw      t7,0xBFE00000
        nop
        li      t2,0xDEADBEEF
        nop
        lw      t2,0xBFE00014
        nop
        andi    t2,t2,0x00000060
        nop
        bne     t2,t4,xt
        nop
        addi    t7,t7,32
        nop
        sw      t7,0xBFE00000
        nop



        nop
        b       lp              /*  branch back to lp  */
        nop

_exit:
        b       _exit
        .end start
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D PART SELECTION

The following tables give a list of compatible radiation hardened and SEU tested

devices. The primary purpose of these tables is to assist in part selection for a space

flight ready TMR design.

SUE TESTED COTS DEVICES

CURRENT PART NUMBER	DESCRIPTION	NEW PART NUMBER	TEST DATE	LET
AT17C512-10JC	SERIAL EEPROM	NONE FOUND		
74ACTC512-10JC	16 BIT BUS TRCVER	74FCT163245	1995	25.2 MEV
ATF22V10C-5JC	EE PLD	CYPRESS 22V10	1993	>120 MeV
AT28C010-15PC	EE PROM	HITACHI HN58C1001	1995	90 MEV
IDT71024S15Y	SRAM 128K X 8 – QTY 20	HITACHI 628128	1996	26 MEV
		MOSAIC MSM8128K	1976	
74AHCT573N	OCTAL TRANSPARTENT LATCH	NONE FOUND		
SN74197N	BINARY COUNTER			
IDT72220L25TP	FIFO (1024 X 8)	IDT7202	1993	3.5 MEV
		IDT7201 512x9	1995	50 MEV
		IDT7203ERP 2048x9	1995	11.6 MEV
		IDT7204 4096x9	1995	11.6 MEV
SN74ACHT11N	TRIPLE 3-INPUT AND GATE	54HC11	1988	>120 MEV
74AHCT541N	INVERTING OCTAL BUFFER/DRV	NONE FOUND		
74AHCT540N	NON-INVRTING OCTAL BUFFER/DRV	54HCT244	1988	>180 MEV

Table D.2. COTS Device List

RADIATION HARDENED DEVICES

CURRENT PART NUMBER	DESCRIPTION	NEW PART NUMBER	QTY	UTMC P/N	PRICE EA.
AT17C512-10JC	SERIAL EEPROM	UT28F256 / 45ns	3	5962R9689103QYX (UTMC PROGRAMMING)	$2,625
74ACTC512-10J	16 BIT BUS TRCVER	UT54ACS164245S	2	5962R9858002QXC (FP ONLY)	$407
ATF22V10C-5JCEE PLD		UT22VP10 / 20ns	2	5962R9475406QLX	$1,837
AT28C010-15PC	EE PROM	UT28F256 / 45ns	4	5962R9689103QYX (UTMC PROGRAMMING)	$2,625
IDT71024S15Y	SRAM 128K X 8 – QTY 20	UT7Q512	5	5962D9960601QUX (FP ONLY)	$944
74AHCT573N	OCTAL TRANSPARTENT LATCH	UT54ACTS373	25(min)	5962R9658901QRX	$159
IDT72220L25TP	FIFO (1024 X 8)		15	NONE AVAILABLE	
SN74ACHT11N	TRIPLE 3-INPUT AND GATE	UT54ACTS11	25(min)	5962R9652301QCX	$159
74AHCT541N	INVERTING OCTAL BUFFER/DRV	UT54ACTS540	25(min)	5962R9659301QRX	$159
74AHCT540N	NON-INVRTING OCTAL BUFFER/DRV FPGA	UT54ACTS541 XQ4013XL-HQ240	25(min)	5962R9659501QRX	$159
AT17C512	Serial EEProm	NONE	3		

Table D.1. Radiation Hardened Device List

147

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. Payne, Jr., J. C., "Fault Tolerant Computing Testbed: A Tool for the Analysis of Hardware and Software Fault Handling Techniques," Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1998.

2. Summers, David., "Implementation of a Fault Tolerant Computing Testbed: A tool for the analysis of hardware and software fault handling techniques," Master's Thesis, Naval Postgraduate School, Monterey, CA, June 2000.

3. Groening, S. E., and Whitehouse, K. D., "Application of Fault-Tolerant Computing for Spacecraft Using Commercial-Off-The-Shelf Microprocessors," Master's Thesis, Naval Postgraduate School, Monterey, CA, June 2000.

4. Olsen, R. C. and Cleary, D. D., Introduction to the Space Environment, class notes for PH2514. December 1997.

5. National Aeronautics and Space Administration. Solar Cell Array Design Handbook. Vol. 1. Pasadena, CA, 1976.

6. Weste, N. H. E. and Eshraghian, K., Principals of CMOS VLSI Design: A Systems Perspective, 2nd Edition, Addison-Wesley, Menlo Park, CA, 1994.

7. Savio, Chau. "Experience of using COTS Components for Deep Space Missions," Proceedings, 4th IEEE International Symposium on, 1999.

8. United Sates. Congress Senate. Committee on Commerce, Science, and Transportation. Subcommittee on Science, Technology, and Space. Decline of the U.S. electronics industry, U.S. G.P.O., 1990.

9. Shirvani, Philip P. and McCluskey, Edward J., Fault-Tolerant Systems in a Space Environment: The CRC Argos Project. Standford University, December 1998.

10. Johnson, B. W., Design and Analysis of Fault Tolerant Digital Systems, Addison-Wesley, New York, NY, 1989.

11. ATMEL Corporation. Online, Internet, Available: www.atmel.com/atmel/faqs/962149423.htm

12. Integrated Device Technology, Inc., The IDT79R3071, IDT79R3081 RISController Hardware User's Manual, Revision 2, Santa Clara, CA, 1994.

13. Xilinx Website. Online, Internet, Available: support.xilinx.com/xbrf/xbrf014.pdf

14. Messenger, G.C. and Ash, M.S., The Effects of Radiation on Electronics Systems, Van Nostrand Rienhold, New York, NY, 1991.

15. Shaeffer, D.L. Kimbrough J.R., Wilburn J.W., Denton, S.M., Kaschmitter, J.L., Colella, N.J., Coakely, P.G., and Casteneda, C., "Proton-Induced SEU, Dose Effects, and LEO Performance Predictions for R3000 Microprocessors," IEEE Transactions on Nuclear Science, Vol. 39, No. 6, December 1992.

16. Shaeffer, D.L. Kimbrough J.R., Wilburn J.W., Denton, Shih, D., J.L., Colella, N.J., Coakely, P.G., Koga, R., Clark, D.A., Ullmann, J.L. and Casteneda, C., "Single Event Effects and Performance Predictions for Space Applications of RISC Processors," IEEE Transactions on Nuclear Science, Vol. 41, No. 6, December 1994.

17. Xilinx, The Programmable Logic Data Book 1999, Xilinx Inc., 1999.

BIBLIOGRAPHY

1. Texas Instruments, Inc., "TL16C750 Asynchronous Communications Element with 64-Byte FIFOs and Autoflow Control," Online, Internet, Texas Instruments, Inc.,

2. Maxim Integrated Products, "MAXIM +5V-Powered, Multichannel RS-232 Drivers/Receivers," Online, Internet,
   Available: http://pdfserv.maxim-ic.com/arpdf/1798.pdf

3. Atmel Corporation, "High Performance EE PLD: ATF22V10C," Online, Internet,
   Available: http://www.atmel.com/atmel/acrobat/doc0735.pdf

4. Logical Devices, Inc., CUPL: Universal Compiler for Programmable Logic User Guide, 1991.

5. Wakerly, J. F., Digital Design: Principals and Practices, 3rd ed., Prentice Hall, Upper Saddle River, NJ, 2000.

6. Sternheim, Eliezer. Singh, Rajvir, and Trivedi, Yatin., Digital Design with Verilog HDL, Cupertino, CA ,1990.

7. Van den Bout, David., The Practical Xilinx Designer Lab Book, Upper Saddle River, NJ 1999.

8. Farquhar, Erin and Bunce, Philip., The MIPS Programmer's Handbook, San Francisco, CA 1994.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1.   Defense Technical Information Center...2
     8725 John J. Kingman Road, Suite 0944
     Ft. Belvoir, VA 22060-6218

2.   Dudley Knox Library...2
     Naval Postgraduate School
     411 Dyer Road
     Monterey, CA 93943-5101

3.   Chairman, Code EC 1
     Department of Electrical and Computer Engineering
     Naval Postgraduate School
     Monterey, CA 93943-5121

4.   Professor Herschel Loomis Code EC/Lm...2
     Department of Electrical and Computer Engineering
     Naval Postgraduate School
     Monterey, CA 93943-5121

5.   Professor Alan Ross Code SP/Ra...2
     Space Systems Academic Group
     Naval Postgraduate School
     Monterey, CA 93943-5110

6.   Captain David C. Summers, USMC 1
     300 Ammunition Avenue
     Odenton, MD 21113

7.   Lieutenant Damen Hofheinz, USN 1
     8322 Watchtower
     San Antonio, TX 78250