

IMPLEMENTATION OF A RADIATION-TOLERANT COMPUTER
BASED ON A LEON3 ARCHITECTURE

by

David Lee Douglas Turner

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Electrical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

July, 2015

©COPYRIGHT

by

David Lee Douglas Turner

2015

All Rights Reserved

ACKNOWLEDGEMENTS

I would like to thank my parents for their continual support and for always being there for me, my advisor Brock LaMeres for his guidance and support, and my labmates Sam Harkness and Justin Hogan for their help and advice. I would like to especially thank Ray Weber, the wizard of FPGA design. Without his assistance this project may never have been completed at all.

TABLE OF CONTENTS

1. BACKGROUND AND MOTIVATION	1
Introduction	1
Computing Requirements of Future Space Missions	2
The Benefits of the LEON3 Processor	3
The Problem of Radiation in Space	3
Sources of Ionizing Radiation	5
Causes of Variations in Radiation Levels	7
Types of Radiation-induced Damage	9
2. TRADITIONAL SOLUTIONS FOR RADIATION	12
Shielding	12
Circuit Level Solutions	13
3. EMERGING SOLUTIONS FOR RADIATION.....	17
FPGAs and Triple Modular Redundancy.....	17
Partial Reconfiguration and Scrubbing	18
Custom FPGAs with Standard Processors.....	20
Standard FPGAs with Custom Processors.....	21
3. MSU'S APPROACH TO RADIATION TOLERANCE.....	23
Previous Work with MicroBlaze and Virtex-6	23
Current Fault Radiation Tolerance Software	29
Fault Tolerance Test Flights.....	31
Transition to LEON3 and Artix-7	34
The Artix-7 FPGA	35
4. FAULT TOLERANCE APPROACH FOR A LEON3 ARCHITECTURE.....	36
The LEON3 Architecture.....	36
Cobham Gaisler Components.....	38
LEON Implementation Instructions	39
5. EXPERIMENTAL RESULTS	41
Single Core LEON.....	42
Four Core LEON	44
Four Core, Reconfigurable LEON.....	46

TABLE OF CONTENTS – CONTINUED

Four Core, Reconfigurable LEON with Triple Modular Redundancy	49
Future Four Tile, Reconfigurable Design	55
FPGA Resource Usage and Configuration Timing	55
Commentary on Partial Reconfiguration and Resource Usage	57
6. FUTURE WORK	58
REFERENCES CITED	60
APPENDICES.....	63
APPENDIX A: Detailed Instructions for LEON.....	64
APPENDIX B: Additional Instructions for LEON3 on Virtex-6	79
APPENDIX C: Quad-core LEON (TMR with Spare) VHDL Code	85

LIST OF TABLES

Table	Page
1. A listing of the major types of radiation and their energies [10]	6
2. A listing of the different systems implemented in this project and their various characteristics (*estimated).....	56

LIST OF FIGURES

Figure	Page
1. An example of an ionizing radiation strike that can cause corruption of data in a space-borne computer system [6].....	4
2. Approximate locations of the inner (red) and outer (blue) Van Allen Radiation Belts around Earth. These belts protect electronics on Earth from most of the dangerous radiation found in space. Image Credit: NASA. [7]	8
3. Depiction of ion strike at a p-n junction creating separated charge carriers that, depending on other conditions, could cause one of the various types of Single Event Effects [8].....	10
4. Ionizing dose vs. Aluminum shielding thickness. Total dose cannot practically be reduced below a certain level, no matter how much shielding is added [16].	12
5. Example of radiation hardening by design, wherein a “guard ring” is placed around part of the transistor [9].....	14
6. Although they also follow Moore’s Law, radiation-hardened processors lag approximately ten years behind general purpose processors in terms of performance [18].....	15
7. Basic setup of a triple-modular-redundant RHBA design [10].....	16
8. Introducing repair of faulted processor (green dots) greatly increases reliability over simple triple modular redundancy (blue line) [10].....	19
9. Virtex-5QV FPGA, the best rad-hard FPGA that money can buy [11].	20
10. Hardware stack used by the MSU research team [10].....	24
11. Exploded CAD view of the MSU hardware stack [17].....	25
12. Block diagram of the MSU hardware stack [17].....	26
13. FPGA board containing the main and control FPGAs used on previous flights [17].....	27
14. Radiation sensor and board developed by MSU [10].....	29

LIST OF FIGURES – CONTINUED

Figure	Page
15. The nine-tile TMR setup currently employed by the MSU system [10].	30
16. NASA-funded sounding rocket that sent the MSU radiation tolerant computer to space in October 2014 [20].	32
17. Sample screenshot of the GUI used to view MSU flight data.	33
18. An AC701 development board identical to the one used for this project [12].	35
19. Block diagram of a full LEON3 processor including peripherals [14].	36
20. Diagram of a LEON3 CPU [14].	37
21. Block diagram of a single core LEON processor.	42
22. Resource usage of a single core LEON processor on Artix-7 FPGA.	42
23. Implemented single core LEON processor on Artix-7 FPGA.	43
24. Block diagram of a four-core LEON processor.	44
25. Resource usage of a four-core LEON processor on Artix-7 FPGA.	44
26. Implemented four-core LEON processor on Artix-7 FPGA.	45
27. Block diagram of a four-core, partially reconfigurable LEON processor.	46
28. Resource usage of a four-core, partially reconfigurable LEON processor on Artix-7 FPGA.	47
29. Implemented four-core, partially reconfigurable LEON processor on Artix-7 FPGA.	48
30. Diagram of modified LEON3 with voter and TMR+spare architecture.	49
31. Resource usage of a four-core, partially reconfigurable LEON processor with triple modular redundancy on Artix-7 FPGA.	51

LIST OF FIGURES – CONTINUED

Figure	Page
32. Implemented four-core, reconfigurable LEON with TMR on Artix-7 board.....	52
33. Resource usage of a single reconfigurable CPU occupying two clock regions on Artix-7 FPGA.....	53
34. Upon interruption, all four cores of LEON are shown to be executing the same instruction.....	54

ABSTRACT

It is desired to create an inexpensive, open-source, radiation-tolerant computer for space applications using commercial, off-the-shelf parts and a proven space-grade processor. Building upon previous work to develop the triplicate architecture using MicroBlaze soft-processors, this implementation, using a modification of the popular open-source space-grade LEON3 soft processor from Cobham Gaisler, enables more compatibility with NASA and existing space computing resources. A partially reconfigurable, triple modular redundant LEON3 processor was successfully implemented in a four-core design on an Artix-7 Field Programmable Gate Array to demonstrate an inexpensive and open-source method of developing radiation-hardened-by-architecture computer systems.

BACKGROUND AND MOTIVATION

Introduction

As the computing requirements of future space missions grow, it has become increasingly enticing to meet these demands with a solution that is inexpensive, easily modifiable, and which provides increased computation. Traditionally, custom radiation-hardened Application-Specific Integrated Circuits (ASICs) have been used for space-borne computers, but these have three significant problems. First of all, low production volumes leads to them being very expensive. Second of all, the physical processes used to make them radiation-hardened tend to lag many years behind cutting-edge computing advancements, which greatly limits their computing power. Third, the reliance on ASICs means that it is much more difficult and time-consuming to upgrade and modify the computer. A less expensive and more powerful computer with a shorter lead time is the ideal solution to this problem.

This thesis presents a solution in the form of a radiation-tolerant architecture running the LEON3 soft-core processor on a Field Programmable Gate Array (FPGA). Solutions of this form do currently exist, but require either expensive custom-made hardware or expensive proprietary software. The LEON3 processor, on the other hand, is an open-source SPARC V8-compliant soft-core processor created by the European Space Agency and further developed and maintained by Cobham Gaisler, a European-based software company. Already used in several space missions, LEON is an ideal soft processor for creating a radiation-hardened FPGA design. A radiation-hardened version,

LEON3FT, can be purchased from Cobham Gaisler, but is expensive and not open-source. Therefore, it was decided to take the open-source version, LEON3, and modify it to make a low cost, radiation-tolerant soft processor.

Computing Requirements of Future Space Missions

Modern day space missions carry a host of sensors and payloads that can require substantial computing resources. For example, a SpaceX Dragon capsule carries as many as 54 separate processors. This includes 18 different units, each with three processors to increase reliability. To save cost, these processors are not individually radiation-hardened, and therefore have to be used in sets of three, which drastically increases the number of processors in use [21]. Another reason to increase computing resources on spacecraft is to perform more data analysis onboard. This allows less data to be streamed back to the ground for human analysis, increasing the potential scientific and engineering benefits of a mission without requiring more downlink bandwidth [17].

Furthermore, as spaceflight moves toward becoming reusable and inexpensive, there will be additional pressure to create less expensive payloads that can use off-the-shelf parts and have a shorter lead time for upgrades. This creates an ideal environment for the use of FPGAs, on which fault tolerance can be implemented inexpensively, and which can be modified quickly and without the need for new hardware. Unlike ASICs, FPGAs can be used to create fault-tolerant computers in an off-the-shelf configuration, wherein inexpensive, mass-produced hardware is used and the only modifications necessary are software-related. This allows for more modern, high-performance

hardware to be used, thereby increasing the computational abilities without a loss of fault tolerance.

The Benefits of the LEON3 Processor

Once the appropriate hardware paradigm has been chosen, the next consideration is choosing a high quality processor that has been proven in the space environment, and which can be modified in order to implement sufficient fault tolerance to resist the effects of radiation. The LEON3 processor, produced by Cobham Gaisler, is the focus of this thesis.

Although the radiation-hardened fault-tolerant version called the LEON3FT is expensive, the standard LEON3 version is open-source. The open-source version lacks built-in radiation-hardening or TMR architecture; however these features can potentially be implemented by the end-user. Due to the existing popularity of LEON for space missions, it is a proven technology for this application. By applying an existing fault mitigation approach developed at MSU to the LEON3, a low-cost equivalent to the LEON3FT can be produced.

The Problem of Radiation in Space

The need for specialized computers for space-borne applications stems from the fact that there are much higher levels of ionizing radiation in outer space than are found on earth. This is because the earth is protected by its magnetosphere, which deflects or traps most of the harmful particles, as well as the earth's atmosphere, which attenuates

the energy of radiation that is not deflected before it reaches the earth's surface.

However, once the surface of the earth is left behind, several different sources of ionizing radiation must be taken into account.

To interfere with the proper functioning of an electronic device, a source of radiation must be ionizing. This means that it either has enough energy to change the orbits of electrons, causing electron/hole pairs and resulting in excess charge in the circuit with the potential to alter proper electrical operation, or to break bonds, which may create permanent damage to the device structure. There are numerous types of particles, as well as electromagnetic radiation, with enough energy to have these effects. An example illustration of a radiation strike is shown in Figure 1:

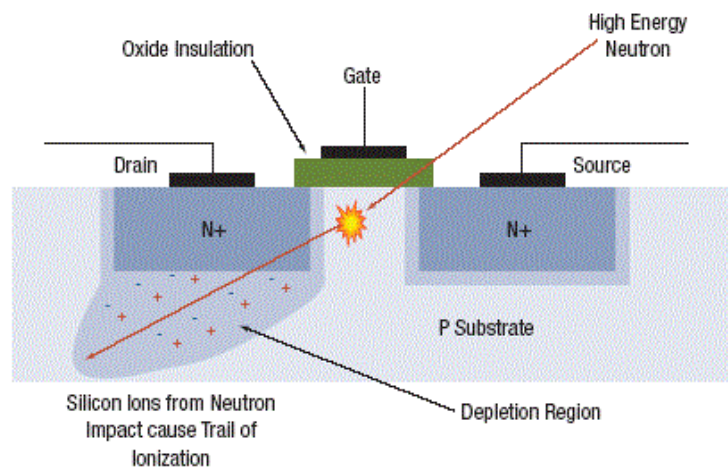


Figure 1. An example of an ionizing radiation strike that can cause corruption of data in a space-borne computer system [6].

Before detailing the different types of radiation, it is useful to cover a few important units and terms that are used to describe the intensity of radiation and the

durability of electronic components. An electron-volt is the most common term used to describe the energy levels of ionizing radiation. An electron-volt is equal to the amount of energy required to raise a single electron through a potential difference of one volt, or $1 \text{ eV} = 1.6 \times 10^{-19} \text{ J}$. In general, to be considered ionizing, a radiation source must have an energy of at least 10 electron-volts (eV).

The other important measurement of radiation is the absorbed dose, which is the total amount of ionizing energy that the device has been exposed to per unit mass. Absorbed dose has units of rads, which is defined as $1 \text{ rad} = \text{an absorbed radiation dose of } .01 \text{ J/kg}$ [1].

The various particles that make up ionizing radiation come in different energy levels with corresponding amounts of damage potential, depending on their source. The least energetic particles are those trapped by earth's magnetosphere and the Van Allen Radiation belts. Although these particles are less energetic, they tend to be more numerous. Other particles move faster because they are part of the solar wind, and come from the sun. But the most energetic ionizing radiation comes from interstellar sources and tends to be travelling extremely fast.

Sources of Ionizing Radiation

The main sources of ionizing radiation are high-frequency electromagnetic waves, alpha particles, beta particles, protons, and neutrons. These sources and their associated energies (which correlate with their potential to cause damage) are listed in Table 1.

Type	Description	Energy
Gamma rays	Very hi-freq EM	
X-rays	Hi-freq EM	
Alpha particles	Helium-4 nuclei	5 MeV
Beta particles	High-speed electrons, outer Van Allen belt	≤ 100 MeV
Trapped protons	inner Van Allen belt	≤ 100 MeV
Solar protons		≤ 1 GeV
Neutrons		
Cosmic rays	Extra-solar origin	TeV

Table 1. A listing of the major types of radiation and their energies [10].

High frequency electromagnetic radiation with enough energy to cause ionization includes x-rays, gamma rays, and some high-energy ultraviolet rays. Gamma rays often come in bursts all the way from other galaxies, and x-rays often have exotic interstellar sources as well. Gamma rays are very difficult to shield against, but are not as harmful as some other forms of radiation, as they are composed of electromagnetic radiation rather than particles.

Alpha particles, which are very energetic and high-speed helium-4 nuclei, are the byproduct of nuclear fusion reactions and make up part of the cosmic rays. Alpha particles are relatively easy to shield against, as they can be stopped by a sheet of paper.

Beta particles are high-speed electrons that are also produced by nuclear fusion in stars. They have more penetrative ability than alpha particles, but can still be stopped by a sheet of aluminum. There are large numbers of these in the outer Van Allen belt.

Another source of ionizing radiation is protons, of which there are many trapped in the inner Van Allen radiation belt. High-energy protons are sometimes produced by the sun as well. Neutrons are another source of radiation. Since they are electrically

neutral, they tend to have less of a direct effect, and instead cause problems through secondary effects.

Cosmic Rays are very high energy particles that mostly originate in violent events outside of the solar system. This is not a specific type of particle but rather describes a category of origin and energy level. Cosmic ray particles have potentially the highest energy of the particles discussed here and therefore can be very dangerous.

Causes of Variations in Radiation Levels

Spaceflights in Low Earth Orbit (LEO) such as those to the International Space Station (about 200-300 miles above the Earth) are largely protected from the radiation problem because most of the radiation is kept away from earth by the Van Allen radiation belts. The lower Van Allen belt is located between approximately 600 and 3,700 miles above the Earth's surface. Once at this altitude, however, the problem can become worse due to particles trapped in the belt. See Figure 2 on the following page.

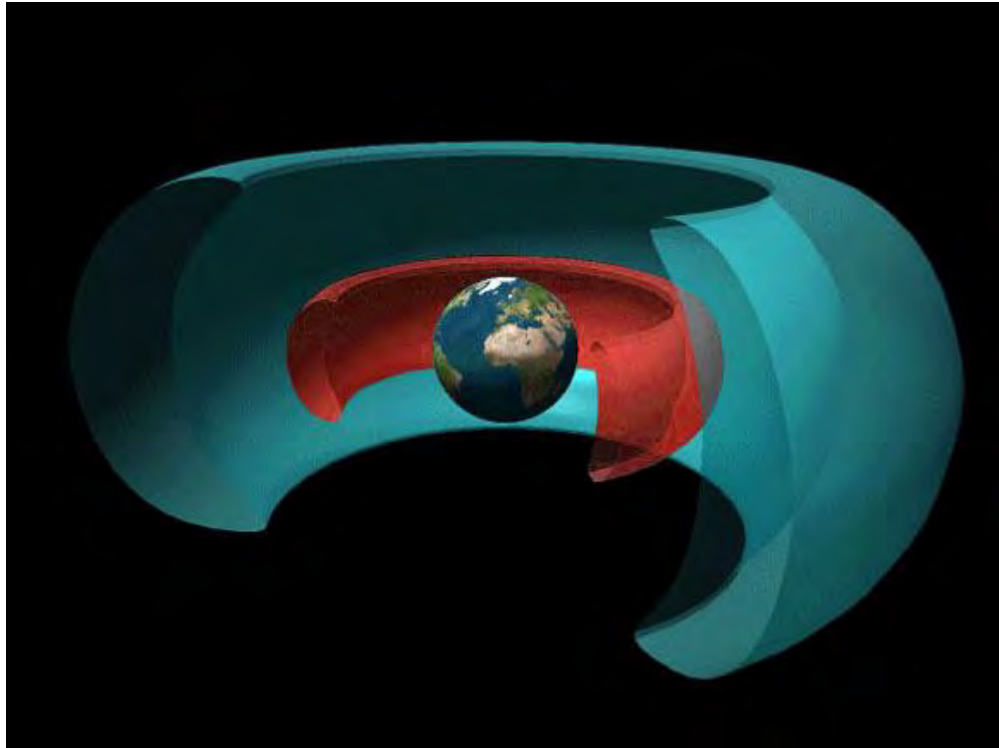


Figure 2. Approximate locations of the inner (red) and outer (blue) Van Allen Radiation Belts around Earth. These belts protect electronics on Earth from most of the dangerous radiation found in space. Image Credit: NASA. [7]

Spaceflights in LEO are not entirely protected from radiation, however, and flight-critical computers require stringent radiation hardening measures even at this low altitude. One major cause of this is the South Atlantic Anomaly, an area located over South America and the southern Atlantic Ocean where the inner Van Allen belt drops down in altitude into the LEO altitude range.

The radiation problem also varies with time, especially as a result of sun cycles & Coronal Mass Ejections. During the peak of the solar cycle, there are more ultraviolet and x-rays emitted by the sun. Coronal mass ejections are enormous eruptions in the sun that occasionally produce large outbursts of protons and electrons.

An interesting note is that designing a radiation-hardened computer system that can survive in space will not enable that computer to function everywhere in the solar system. By way of example, the magnetosphere of Jupiter is so strong that radiation levels there are many times higher than in Earth's Van Allen belts. A radiation-hardened computer system designed to survive in such an environment would need to be much more reliable than one designed for general-purpose space use.

Types of Radiation-induced Damage

Radiation-induced damage is in general divided into two major categories: Total Ionizing Dose, which measures cumulative exposure, and Single Event Effects, which describe problems caused by excess charge created in the circuitry. Single Event Effects are then further divided based on severity. Space-borne computers must be able to handle both kinds of damage, although protection from the two types can vary drastically.

Total Ionizing Dose (TID) is the total absorbed dose of radiation that the electronic part has been exposed to over time. After a certain point, the buildup of radiation-induced damage in the semiconductor can influence the threshold voltages of the transistors and eventually cause some to stay permanently on, creating a short-circuit that can lead to permanent device failure.

The TID a device can absorb before failure is an important measure of radiation resistance. Fortunately, TID immunity tends to increase with smaller gate size technology. TID becomes a problem with an increase in trapped charge in oxide layers, an occurrence which decreases along with the thickness of the oxide. For this reason,

using a modern small gate size technology circuit can actually decrease susceptibility to TID-induced failure.

Single Event Effects (SEEs), on the other hand, are transient voltage fluctuations that are the result of a short-term radiation bombardment rather than a long-term buildup over time. Single Event Effects increase as the gate size decreases and come in different types, ranging in severity from harmless, to causing permanent device failure. An example of an ion strike creating unwanted charge in a silicon device is depicted in Figure 3 below.

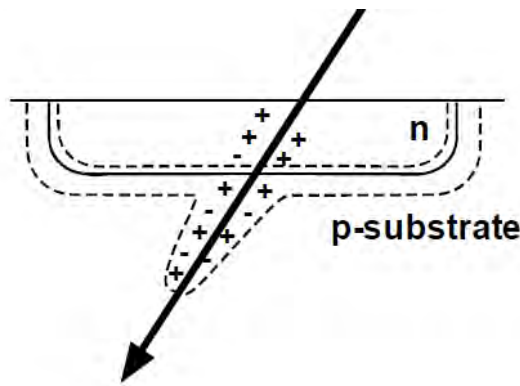


Figure 3. Depiction of ion strike at a p-n junction creating separated charge carriers that, depending on other conditions, could cause one of the various types of Single Event Effects [8].

The three different types of Single Event Effects are described below:

Single Event Transient (SET) – in this event, high-energy radiation interacts with the system and causes a short-term fluctuation in voltage or current. This will not necessarily affect the operation of the circuit, however, unless the voltage fluctuation is

latched into a memory state or is used in a calculation before it dissipates. In this case it becomes a Single Event Upset (SEU).

Single Event Upset (SEU) – if a radiation strike causes a change in voltage that gets latched into memory, then a Single Event Upset is created. Although the computer chip is not permanently damaged, this type of event causes data corruption and requires the affected part of the chip to be scrubbed.

Single Event Functional Interrupt (SEFI) – a severe type of SEU in which the device's control circuit is placed into an illegal 'blocked' condition. This halts normal operation of the device and requires a power reset for recovery.

Single Event Latchup (SEL) – transistors short out, possible permanent damage. This type of event creates a short circuit between power and ground. In some cases the device will simply short out and need to be rebooted. In more extreme cases, however, the power drawn due to the short circuit can permanently damage the device. Therefore, this is the most extreme category of Single Event Effect [1].

TRADITIONAL SOLUTIONS FOR RADIATION

Shielding

The most obvious and simple solution to prevent radiation damage is shielding. Shielding is commonly used in nuclear power plants and other earthbound radiation environments since it is inexpensive and simple. Unfortunately, this is a difficult solution for use in outer space since the cost of launching the weight of the shielding into orbit is high. Furthermore, the effectiveness of shielding follows a logarithmic law, such that a thin piece of shielding material can stop low-energy radiation, but much thicker shielding is needed to stop higher-energy radiation. This effect is demonstrated in Figure 4 below:

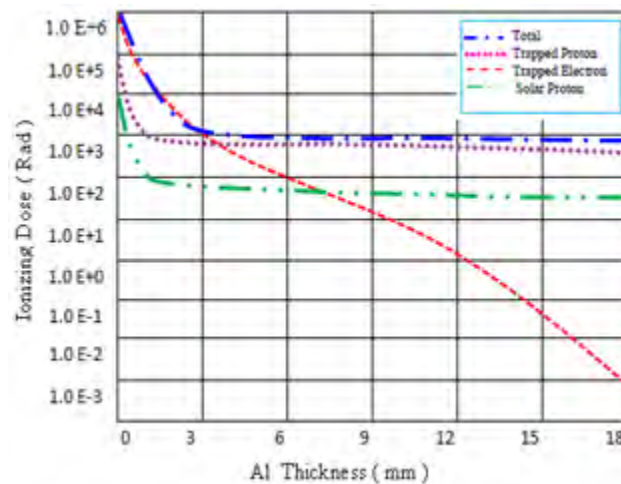


Figure 4. Ionizing dose vs. Aluminum shielding thickness. Total dose cannot practically be reduced below a certain level, no matter how much shielding is added [16].

In order to for the shielding to be effective enough to allow continuous, uninterrupted operation of the underlying computing resources for mission-critical purposes, such a

high mass of shielding may be required as to make this option impractical or very expensive. Furthermore, shielding can sometimes make the radiation problem worse, since high-energy radiation can create secondary radiation particles upon striking the shielding. What is needed, then, is a circuit-level solution which modifies the computer circuit itself to create fault tolerance, therefore adding less weight to the design.

Circuit Level Solutions

There are three main categories of radiation-hardening techniques implemented on the circuit level. Although all three have been used in the past, they vary greatly in terms of cost and lead time to implement.

Radiation hardening by process (RHBP) is the most extensive and expensive means of radiation hardening. This method seeks to decrease the amount of defects present in the material through a carefully-controlled manufacturing process in order to decrease the amount of charge that will be trapped as a result of radiation bombardment. The downside of this technique is that it requires a significant change in the manufacturing setup. Therefore, it is more expensive since the economies of scale produced by standard chip manufacturing cannot be fully leveraged for making RHBP chips. Furthermore, there will naturally be a very long lead time for any improvements in this arena since new manufacturing processes and equipment would be needed.

The next option is radiation hardening by design (RHBD). In this method, the layout of the board is modified to mitigate the effects of radiation strikes by placing barriers between transistors. Two common methods for doing this are Local Oxidation of

Silicon (LOCOS) and Shallow Trench Isolation (STI). This is less expensive than RHBP because it does not require a separate process to produce, and can be manufactured on standard equipment. The downside is that additional chip space is required for the barriers. This extra distance can also potentially limit the clock speed of the circuit. See Figure 5 below for examples of RHBD.

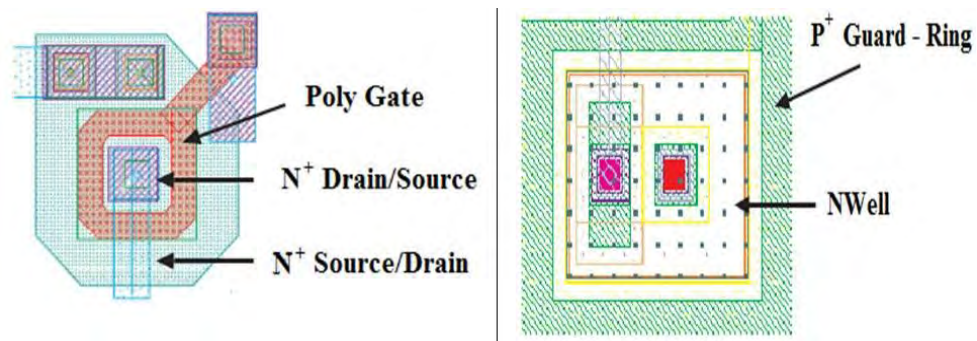
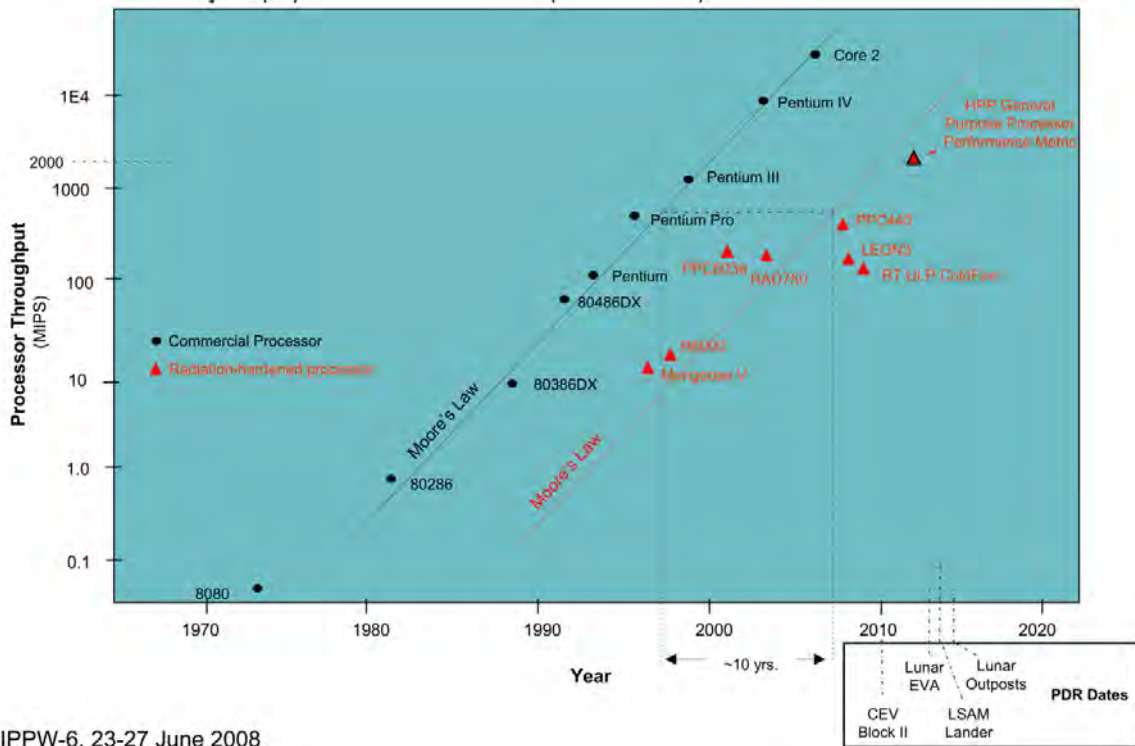


Figure 5. Examples of radiation hardening by design, wherein a “guard ring” is placed around part of the transistor [9].

One of the major downsides of radiation hardening through process and design is long time it takes to develop these technologies, since completely new chip layouts and designs are required for new modifications. For this reason, such chips tend to lag several years behind their commercial, off-the-shelf (COTS), non-radiation-tolerant counterparts in terms of performance. This trend is shown in Figure 6.



IPPW-6, 23-27 June 2008

Figure 6. Although they also follow Moore’s Law, radiation-hardened processors lag approximately ten years behind general purpose processors in terms of performance [18].

To overcome this inherent limitation in hardware development, the technique of radiation hardening by architecture (RHBA) has been developed. In this method, triple modular redundancy (TMR) is used to allow radiation resistance to be programmed into a traditionally manufactured device, at the expense of computing resources. This method is illustrated in Figure 7.

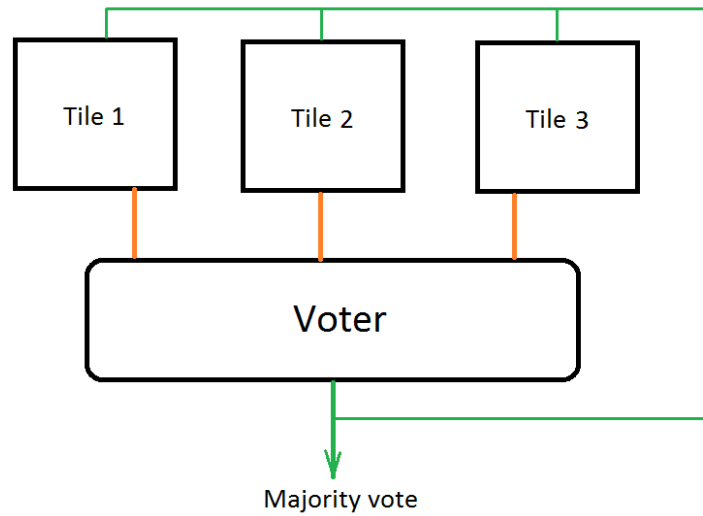


Figure 7. Basic setup of a triple-modular-redundant RHBA design.

Using this technique, three processors run in parallel, and a voter decides on the appropriate output. A radiation strike causing an error in a single processor will be overridden by the uncorrupted outputs of the other two processors. The faulted processor can then be corrected or reset as appropriate. Having a spare processor available (bringing the total to at least four) can dramatically reduce the time necessary to continue with calculations. The faulted processor can then be scrubbed in the background and reintroduced as a spare for future incidents.

EMERGING SOLUTIONS FOR RADIATION

FPGAs and Triple Modular Redundancy

Although Radiation Hardening by Process and Radiation Hardening by Design have many good qualities, they inherently involve hardware modifications, and therefore are more expensive and time-consuming to implement than Radiation Hardening by Architecture. Radiation hardening by Architecture, then, is the method of choice when an inexpensive, quick, and yet still reliable method is desired. The benefits of using this method are outlined below.

Although FPGAs tend to be more expensive than ASICs in general, radiation-hardened FPGAs can be made much more inexpensively than radiation-hardened ASICs because radiation resistance can be implemented on a Commercial, off-the-shelf (COTS) FPGA using RHBA by simply reprogramming it to implement triple modular redundancy. On the other hand, a radiation tolerant ASIC must be designed and manufactured specifically for that purpose using RHBP or RHBD.

Radiation-hardened ASICs also have the downside of being very difficult to update or modify. Although new instruction code can be uploaded, the functional layout of the device will remain the same, and must generally be finalized months to years before implementation. An FPGA, on the other hand can be reprogrammed to utilize updated functionality even after launch. Use of FPGAs can also greatly reduce lead time for a project, since the hardware can be finalized before the actual FPGA synthesis.

Therefore, the detailed design work to implement the configuration of the FPGA can take place much closer to the deadline since no hardware changes are involved.

FPGAs can be reconfigured upon startup so that the same processing real estate is optimized for different uses at different times. Yet, the FPGA is still more efficient than a traditional computer processor because it can be configured to efficiently perform the task at hand, and therefore is more similar to an ASIC in terms of data throughput, etc. Reconfiguration can also be used to replace or “scrub” the configuration in the FPGA to reset it to a known good state, although reconfiguring the entire chip at once would interrupt its ability to carry out the program.

Partial Reconfiguration and Scrubbing

Taking this a step further, partial reconfiguration allows for the change or reboot of only part of the FPGA processing resource independently of the rest. This allows the faulted portion of the FPGA to be reconfigured and reset while the rest of the FPGA resources continue to function. Partial reconfiguration also takes less time than full reconfiguration since fewer resources need to be written to. This could be a significant advantage in a high-radiation environment, where the time required to bring a spare processor back to a good state can have a large effect on the mean time to failure, as shown in Figure 8.

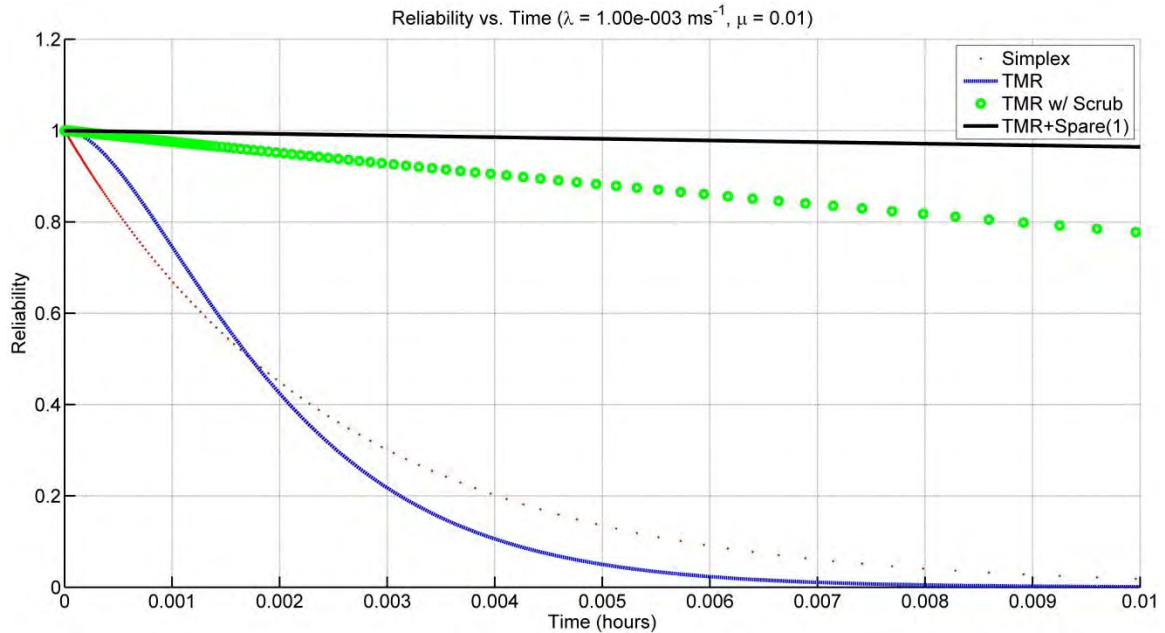


Figure 8. Introducing repair of faulted processor (green dots) greatly increases reliability over simple triple modular redundancy (blue line) [10].

The rewriting process of the FPGA resources is accomplished through “scrubbing”, which can take the form of either blind scrubbing or readback scrubbing. In blind scrubbing, the entirety of the memory contents in the scrubbed region is overwritten whether the individual contents are faulty or not. Readback scrubbing, on the other hand, incorporates the ability to read the memory contents and compare them with a known golden copy to determine whether they need to be overwritten or not. The golden copy can potentially be stored in a medium less susceptible to radiation. Both methods of scrubbing allow for periodic refreshment of the FPGA contents, thereby greatly increasing the reliability of the system since faults can be removed. Readback scrubbing, however, is significantly more challenging to implement than blind scrubbing.

The advantages of using FPGAs for radiation tolerant computing, then, are clear. There are two main design strategies that may be used to implement them: custom FPGAs with standard programming, and standard FPGAs running custom programming.

Custom FPGAs with Standard Processors

There are currently multiple commercially available radiation-tolerant FPGAs on which one can run standard soft-core processors. The Virtex-QV series of boards by Xilinx is the industry leader in this category. A Virtex-5QV is shown below in Figure 9.

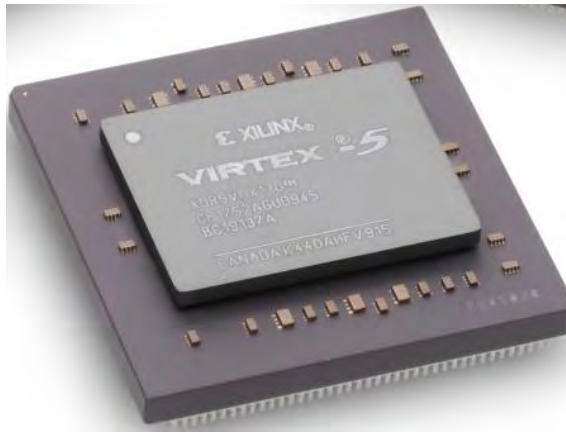


Figure 9. Virtex-5QV FPGA, the best rad-hard FPGA that money can buy [11].

Incorporating both RHBD & RHBA features, the Virtex-5QV series are the top-of-the-line. Unfortunately, they are also expensive and somewhat difficult to obtain. With over 130,000 logic cells, this board has a TID capability of 1 Mrad and a SEL Immunity of $125 \text{ MeV-cm}^2/\text{mg}$. These boards are designed such that any standard soft processor can be run on them.

The Virtex-5QV, which is fabricated using 65 nm processes, is radiation hardened by design to guarantee TID up to 1 Mrad and single-event-latch-up (SEL) immunity. Since smaller process size automatically increases TID immunity, more modern 7-Series devices, fabricated using 28 nm processes, should have good TID immunity even without the special processes used to manufacture the Virtex-5QV. SEL immunity in the Virtex-5QV is enabled using a thin epitaxial layer in the wafer manufacturing process [5].

The Virtex-5QV chips are also designed with built-in Triple Modular Redundancy capability, so that the user's software is automatically triplicated and the output sent through a majority voter circuit. They also include Single Event Upset (SEU) correction.

The Virtex-5QV has been extensively tested and has already flown in space on the Sandia National Labs Materials International Space Station Experiment (MISSE-8) starting in 2011. During this experiment, the Virtex-5QV did not experience any single-event upset errors [2]. The downside is that the Virtex-5QV is very expensive and somewhat difficult to obtain. For example, it is not available from the popular online electronics supplier, DigiKey. It may also be subject to ITAR regulations.

Standard FPGAs with Custom Processors

Conversely, there are also radiation-resistant soft processors available that can be run on any standard FPGA. One of these is the LEON processor – a soft processor created by the European Space Agency and used in many space missions.

LEON is a 32-bit soft-core processor based on SPARC-V8 architecture. It was originally created by the European Space Agency in 1997 and later further developed by

Gaisler Research. There have been four versions released so far, with the most recent radiation-tolerant version being the LEON3FT. The LEON3FT offers built in fault tolerance using TMR and automatic error correction, but costs a good amount of money.

The LEON3FT can be implemented on both radiation-tolerant and conventional FPGAs. This can be a distinct advantage if there is any difficulty in procuring a radiation-tolerant FPGA.

The LEON processor is also a proven technology with a track record of successful space-borne implementation. The fault-tolerant version of the LEON2 processor has successfully been used in numerous space missions, including ESA's Alphasat telecom satellite, the Proba-V microsatellite, the Earth-monitoring Sentinel family and the upcoming BepiColombo mission to Mercury [3].

Another advantage of the LEON processor is that the non-radiation-tolerant version is open source and can be freely downloaded from the Cobham Gaisler website. Furthermore, the processor can be run on any brand of FPGA. This means that the LEON processor can be freely modified and ported to different platforms.

MSU'S APPROACH TO RADIATION TOLERANCE

Previous Work with MicroBlaze and Virtex-6

The team at Montana State University, Bozeman currently implements triple modular redundancy on a Xilinx Virtex-6 FPGA using Xilinx MicroBlaze soft processors. Nine MicroBlaze processors are placed on the FPGA, with three being active at any given time. This allows for a large number of spare processors, a potential advantage in a high-radiation environment. Additionally, a two-dimensional grid of radiation sensors is placed above the FPGA board to provide information about the environment over time, and also to help correlate any errors with their possible cause.

A great deal of research and design has been done by prior graduate students at MSU to design the hardware stack used to support the main FPGA in spaceflights. The MSU-designed hardware has flown in space multiple times, including on a NASA high altitude balloon and on a sounding rocket. The hardware has been designed to fit into the form factor of a 1U CubeSat to increase the ease of integration and availability of launch opportunities, and has outside dimensions are approximately 4"x4"x4". The CubeSat hardware stack used is shown in Figure 10.

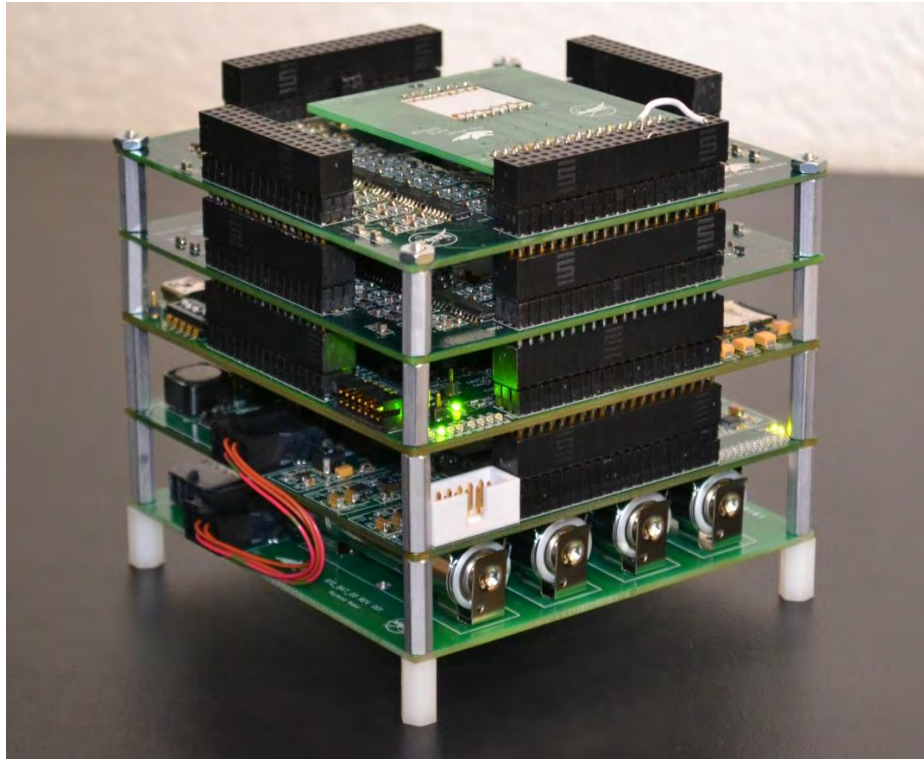


Figure 10. Hardware stack used by the MSU research team [10].

As can be seen, the flight hardware is divided into several separate but interconnected printed circuit boards that together allow the system to function autonomously during flight. Another view is shown in Figure 11, which delineates an exploded CAD model of the hardware stack.

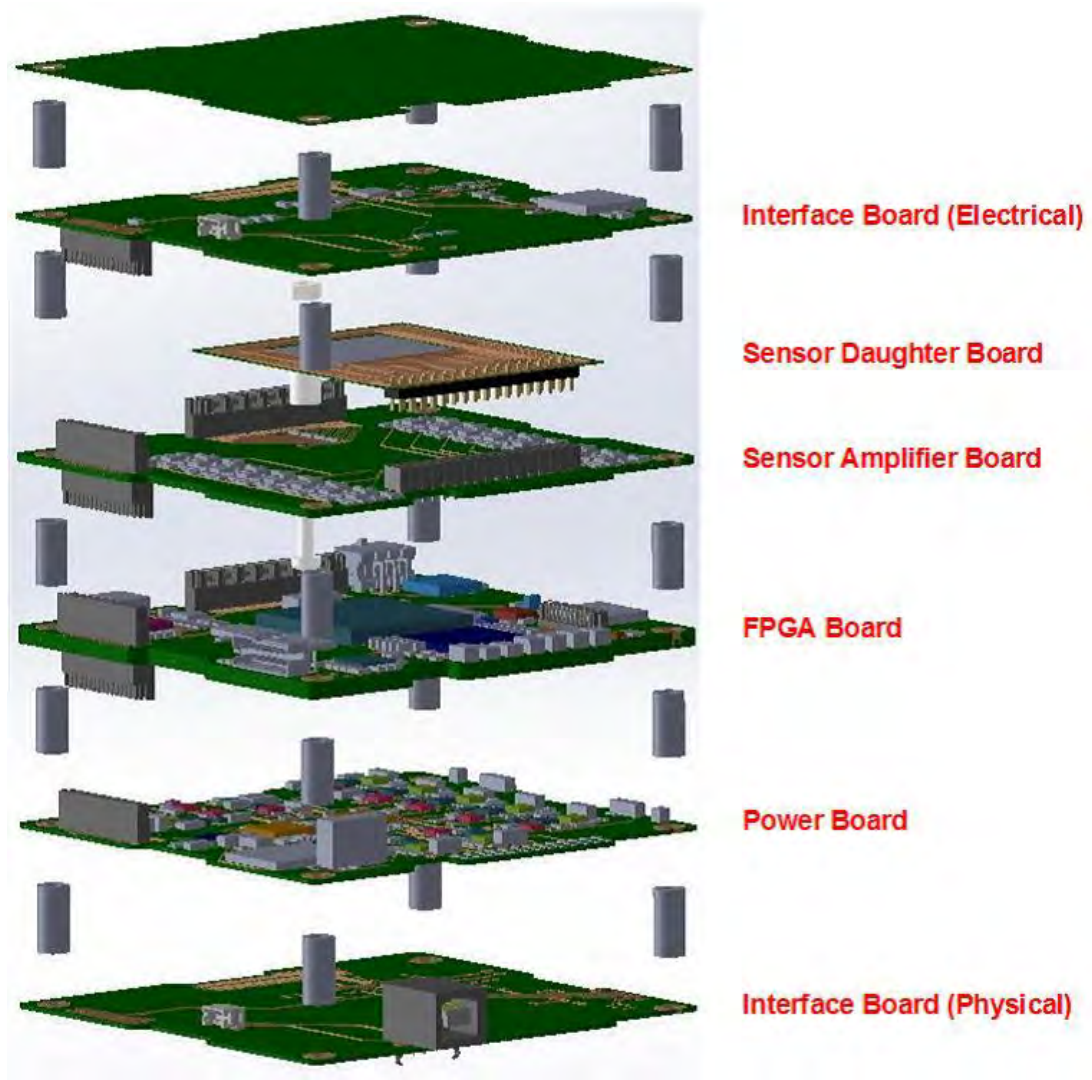


Figure 11. Exploded CAD view of the MSU hardware stack [17].

A block diagram of the overall system is shown in Figure 12.

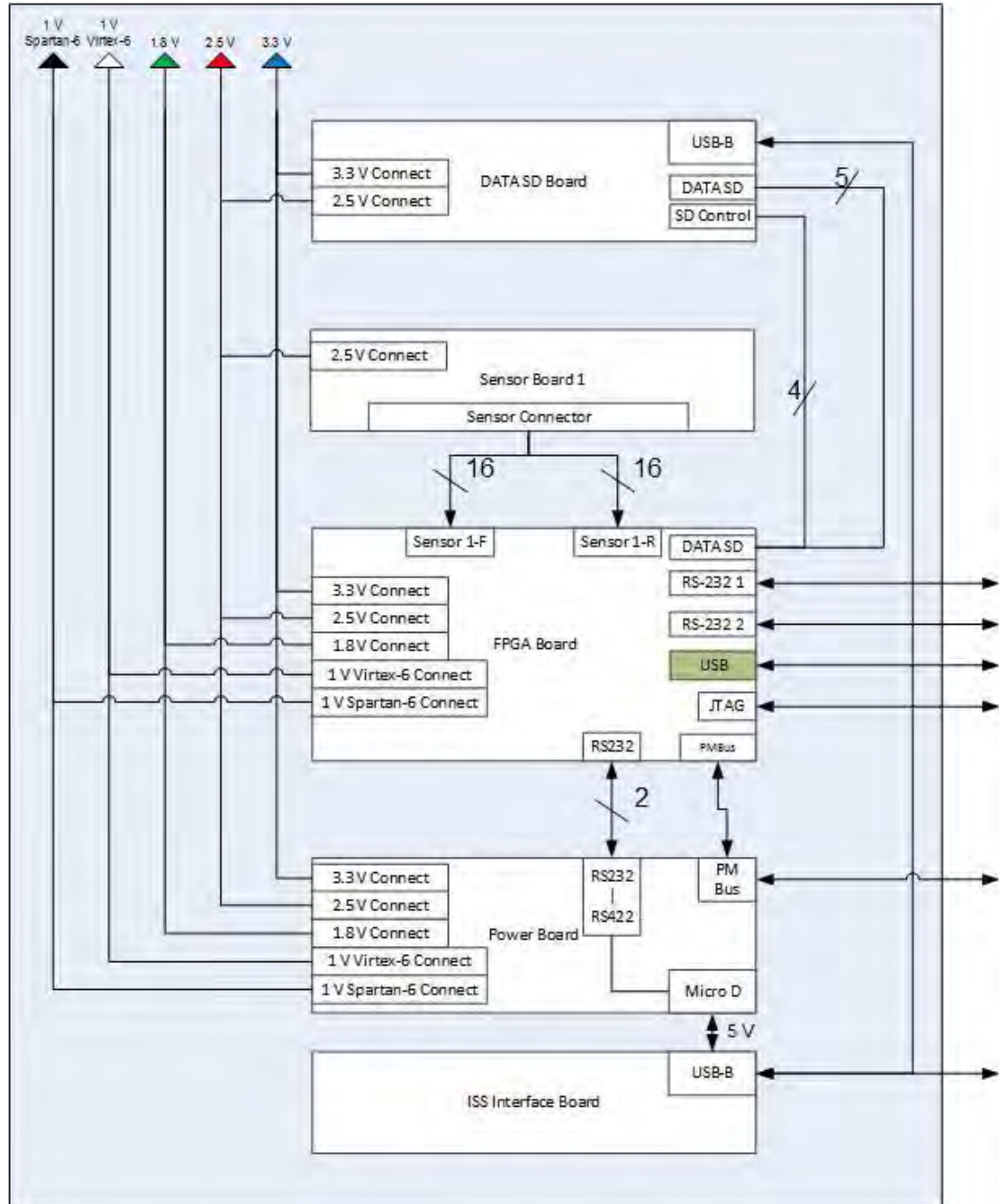


Figure 12. Block diagram of the MSU hardware stack [17].

The first individual part of the hardware stack under consideration is the power board, which must have low power consumption while offering power at several different

voltages. There are currently eleven voltage rails on the power board in both digital and analog configurations. Additionally, the power board incorporates internal monitoring and fault protection [17].

The FPGA board itself has also been the subject of a great deal of effort and redesign. There are two FPGAs located on this board, a main processor FPGA and a smaller control FPGA. The hardware stacks currently available use a Xilinx Virtex-6 for the main processor, although efforts are currently underway to transition to a Xilinx Artix-7. The control FPGA is a Xilinx Spartan-6, and is used to control the functioning of the main FPGA, as well as to write to it the full and partial bitstreams. The board holding the Virtex-6 and Spartan-6 FPGAs is shown in Figure 13 below.

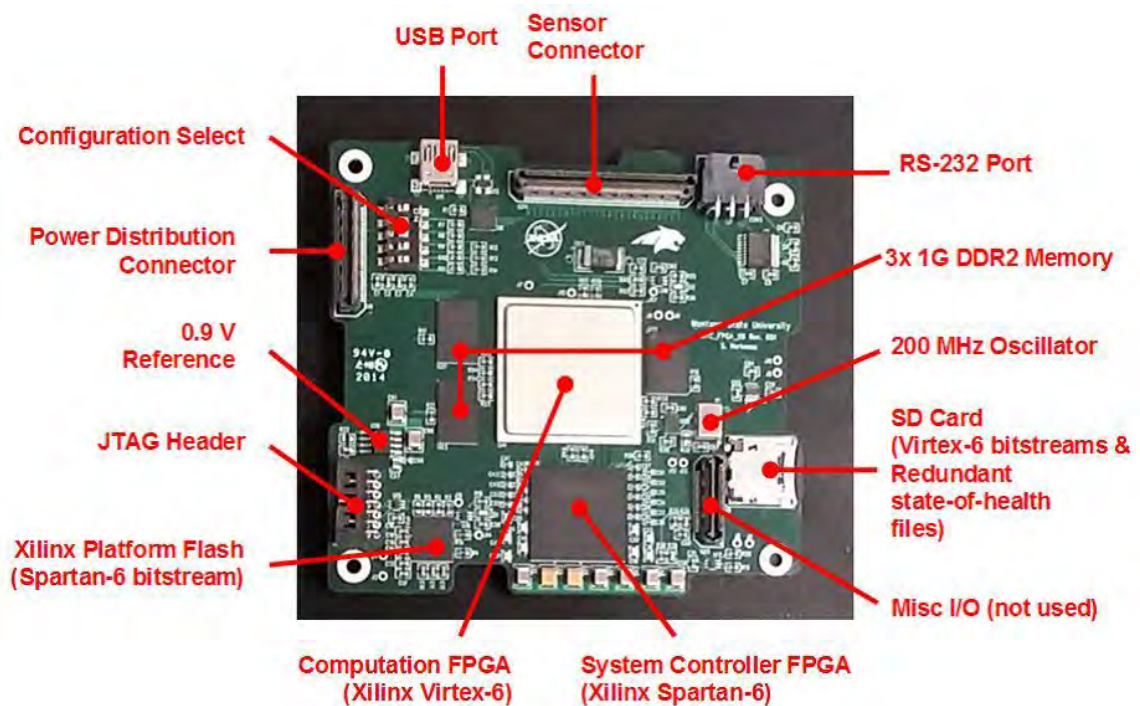


Figure 13. FPGA board containing the main and control FPGAs used on previous flights [17].

If an error is detected in one of the processors running on the Virtex-6, blind scrubbing is used to write over that processor with a partial bitstream in order to return the processor to a clean state.

A MicroSD card is also included in the design as a storage device to hold the bitstream files as well as data collected during operation. Reading from and writing to the MicroSD card is controlled from the Spartan-6. Three 1 GB DDR2 RAM chips are also included on the FPGA board but are not currently used. In the future, however, these could be used to store larger programs and data, for example to enable the running of a Linux operating system on the processors. The system can be configured either through a complete bitstream loaded onto the Xilinx Platform Flash, which enables autonomous operation during spaceflight, or through a JTAG port, which is useful for debugging and system development [17].

A custom radiation sensor and accompanying circuit board were designed and built by the MSU team in order to gather information during flight about the radiation environment. The radiation sensor circuit board is shown in Figure 14.

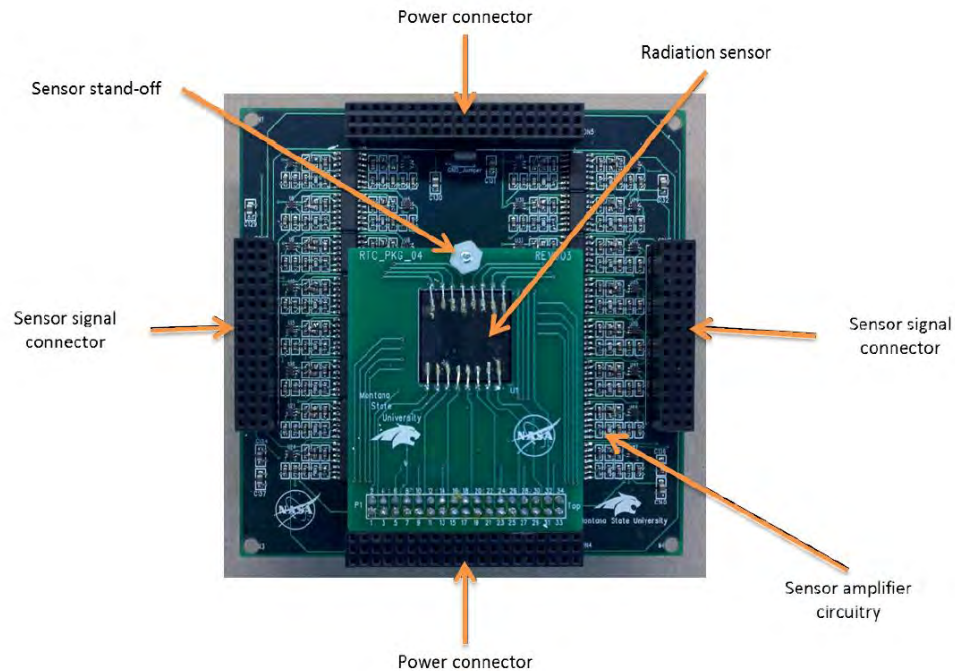


Figure 14. Radiation sensor and board developed by MSU [10].

The initial development of the radiation sensor contains 16 front-side channels and 16 back-side channels arranged perpendicular to each other, giving a total spatial resolution of 256 pixels [10].

Current Fault Radiation Tolerance Software

The current nine-tile TMR+spare architecture with partial reconfiguration runs on the main FPGA and has been developed over the last few years by several different graduate students. The MicroBlaze processors currently in use are 32-bit Harvard architecture soft-core processors. Nine MicroBlaze processors are placed on the main

FPGA, with three being active at any one time and up to six being held in reserve as spares. The setup of this system is illustrated in Figure 15 below.

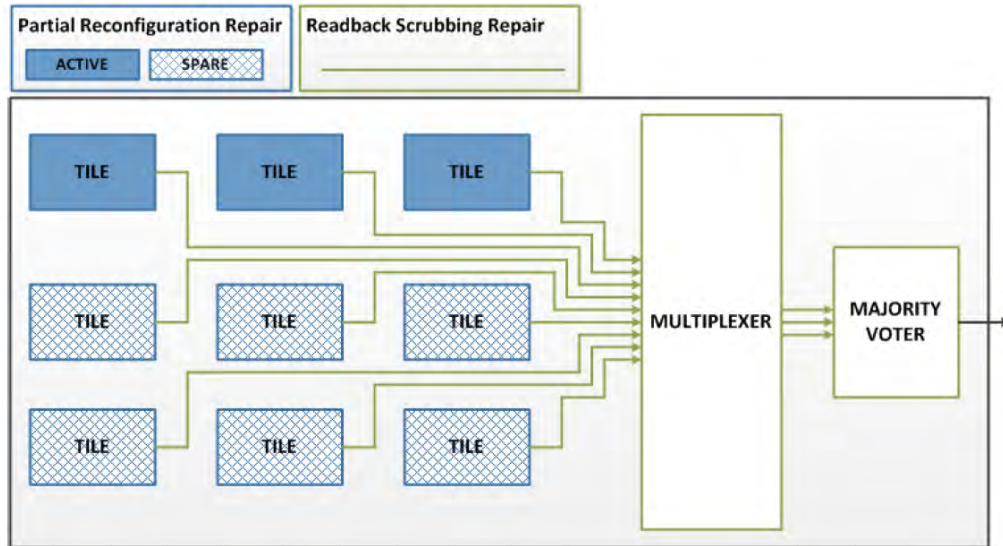


Figure 15. The nine-tile TMR setup currently employed by the MSU system [10].

An important aspect of this system is the partial reconfiguration ability, which allows for the correcting of errors during operation and therefore a large improvement of mean time to failure. The only difference between MSU's approach and the system illustrated above is that blind scrubbing is used in place of readback scrubbing to fix faulted tiles.

One limitation of the current fault-tolerant setup is that only the main FPGA is triplicated. Neither the Spartan-6 control FPGA nor the memory holding the bitstreams are not triple modular redundant. Therefore, a radiation strike on the control FPGA or the memory could cause an unrecoverable error. However, once the proof of concept has been implemented on the main FPGA, it should not be conceptually difficult to apply the same setup to the rest of the design. As mentioned, there are three currently unused 1 GB

DDR2 RAM chips on the FPGA board, so TMR memory could be implemented without requiring any hardware changes.

Fault Tolerance Test Flights

The MSU team has flown the system into space multiple times already to test the radiation tolerance of the system in a realistic environment. Initial testing of the system took place at the Texas A&M Radiation Effects Facility in College Station, Texas using a 25 MeV Krypton radiation beam. This test proved that the radiation sensor and software systems worked, but did not produce actual faults in the MicroBlaze processors. The system was next flown on a high-altitude balloon in both 2012 and 2013 to measure its operation in an actual space environment. The first flight helped to uncover design flaws in the system, and the second flight went well, proving that the design was ready for spaceflight. However, no actual radiation strikes were recorded in either case [10].

Most recently, the system was launched as part of the payload of a NASA-funded sounding rocket in October of 2014. The rocket launch is shown in Figure 16 below.



Figure 16. NASA-funded sounding rocket that sent the MSU radiation tolerant computer to space in October 2014 [20].

The system that flew in October 2014 was in space for a few minutes and ran the MicroBlaze and Virtex-6 system described above. Data concerning radiation strikes, system power levels, and program outputs was logged during flight and written to the MicroSD card. This data was then analyzed post flight using a custom-made Graphical User Interface (GUI). A screenshot of this GUI (showing sample rather than actual flight data) is shown in Figure 17.

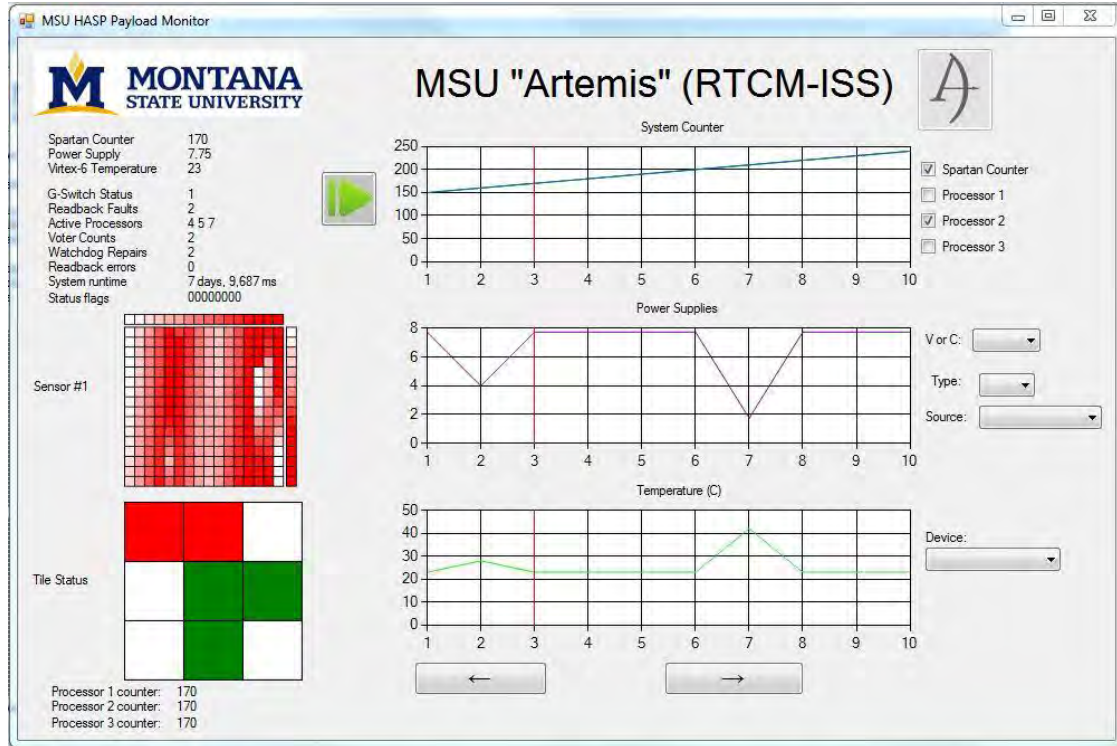


Figure 17. Sample screenshot of the GUI used to view MSU flight data.

Building on the lessons learned from the 2014 flight, the MSU team is updating the system and plans to operate it for several months aboard the International Space Station (ISS) in 2016. Due to limited power available to experiments onboard the ISS, the hardware is being modified to use an Artix-7 FPGA instead of a Virtex-6. Although work is also underway to replace the MicroBlaze processor with the LEON processor, this transition is not expected to be completed in time for the flight to the ISS.

Transition to LEON3 and Artix-7

The MicroBlaze processor works for the present but the next step is to implement a fault-tolerant architecture on a proven space processor. For this work, the LEON3 processor has been chosen. LEON is open-source and can be freely ported to any FPGA platform. The LEON soft processor is furthermore already one of the standards in the space industry, making it a logical next step to implement. By applying MSU's fault-tolerant architecture to the LEON3 to implement triple modular redundancy (TMR), a low-cost near-equivalent to the LEON3FT can be produced.

The LEON3 processor uses a robust and large architecture (compared with the MicroBlaze processor), which is good for running real space missions but makes it more difficult to convert into a reconfigurable design that fits onto an FPGA board. The LEON3 is a 32-bit SPARC version 8 compliant processor, meaning that it uses a reduced instructions set architecture (RISC). It also incorporates a 7-stage pipeline, separate instruction and data caches of the Harvard architecture, an AMBA-2.0 AHB bus interface, and has a high performance of up to 1.4 DMIPS/MHz [15].

Furthermore, MSU is transitioning to using an Artix-7 FPGA in place of the Virtex-6. The two FPGAs have a similar amount of resources available, but the Artix-7 uses far less power, which can make a large difference on power-limited space missions. Furthermore, it allows synthesis and implementation using Xilinx Vivado Design Suite as opposed to the older Xilinx ISE Design Suite. Vivado is more modern and includes a streamlined interface that can greatly increase productivity.

The Artix-7 FPGA

The new FPGA being incorporated by the MSU Research Team is an Artix-7 200T. An AC701 Evaluation Board containing the same FPGA is also in the lab and was used in this project. Such an evaluation board is shown below in Figure 18.

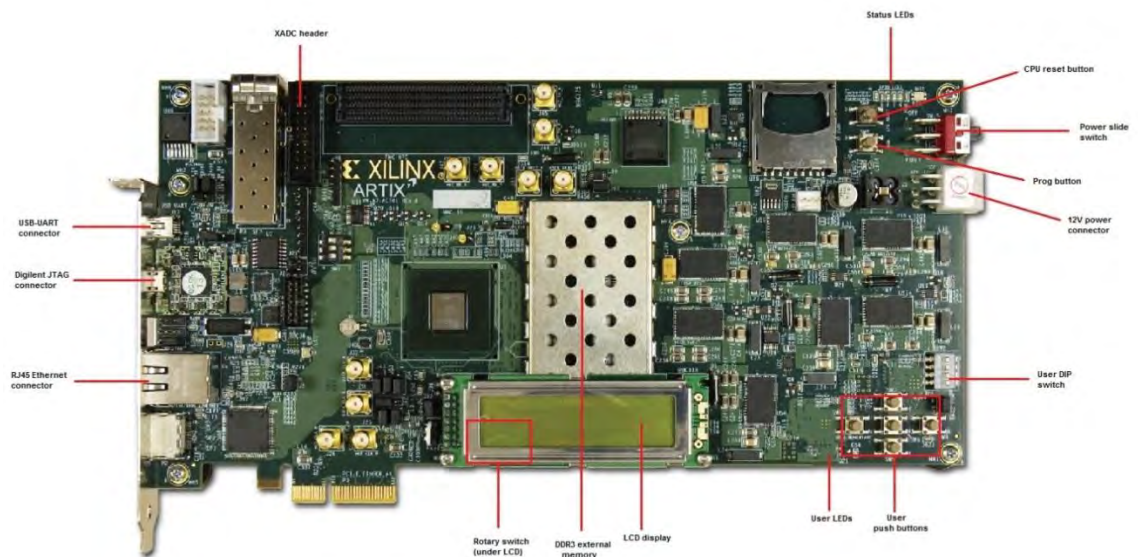


Figure 18. An AC701 development board identical to the one used for this project [12].

The Artix-7 is part of the Xilinx 7 series FPGA family, which was released in 2010 and uses a 28 nm die process. The Artix family is optimized for low power usage and cost. The 200T model, used in the MSU design, is the largest available Artix-7 FPGA. It contains 215,360 logic cells arranged in 33,650 slices, and 13,140 kB of block RAM. Each slice contains four LUTs and eight flip-flops [19].

FAULT TOLERANCE APPROACH FOR A LEON3 ARCHITECTURE

The LEON3 Architecture

A general diagram of the components contained within the LEON3 processor is shown in Figure 19 below.

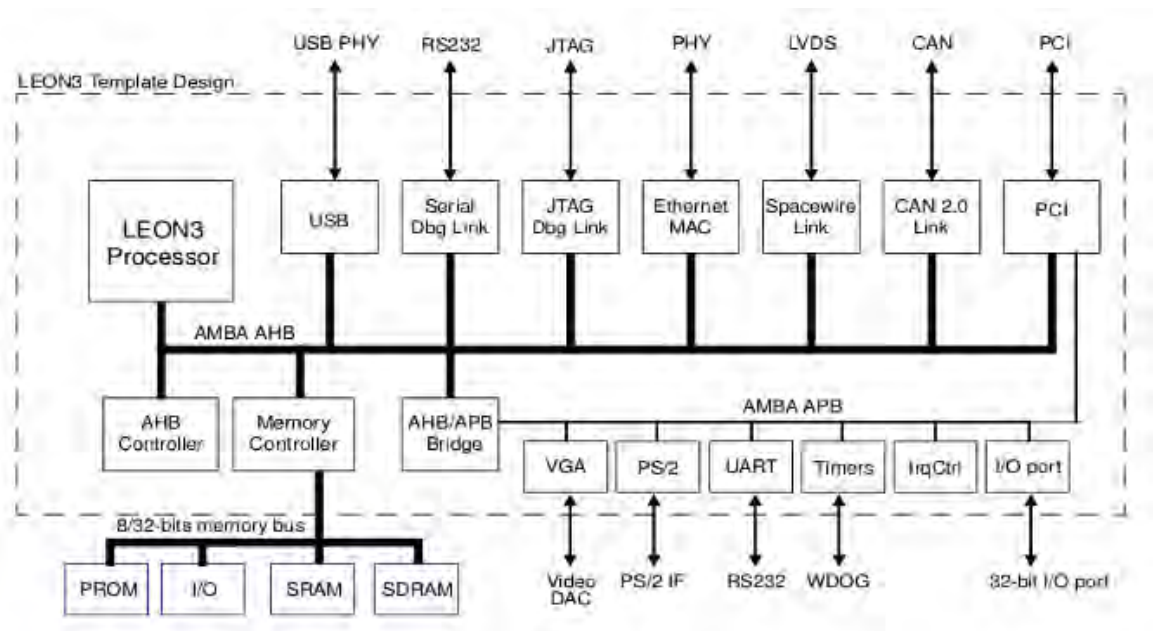


Figure 19. Block diagram of a full LEON3 processor including peripherals [14].

As shown in the above diagram, all of the components of the LEON processor are connected via the AMBA AHB bus. To save space on the FPGA, unneeded components may be turned off. These include the Ethernet MAC and Serial Debug Link. The vital components that must be kept are the JTAG Debug Link, the AHB bus, some on-board SRAM, and four CPUs. All components are connected via the AMBA AHB bus. A more detailed diagram showing only a CPU of the LEON3 is given in Figure 20.

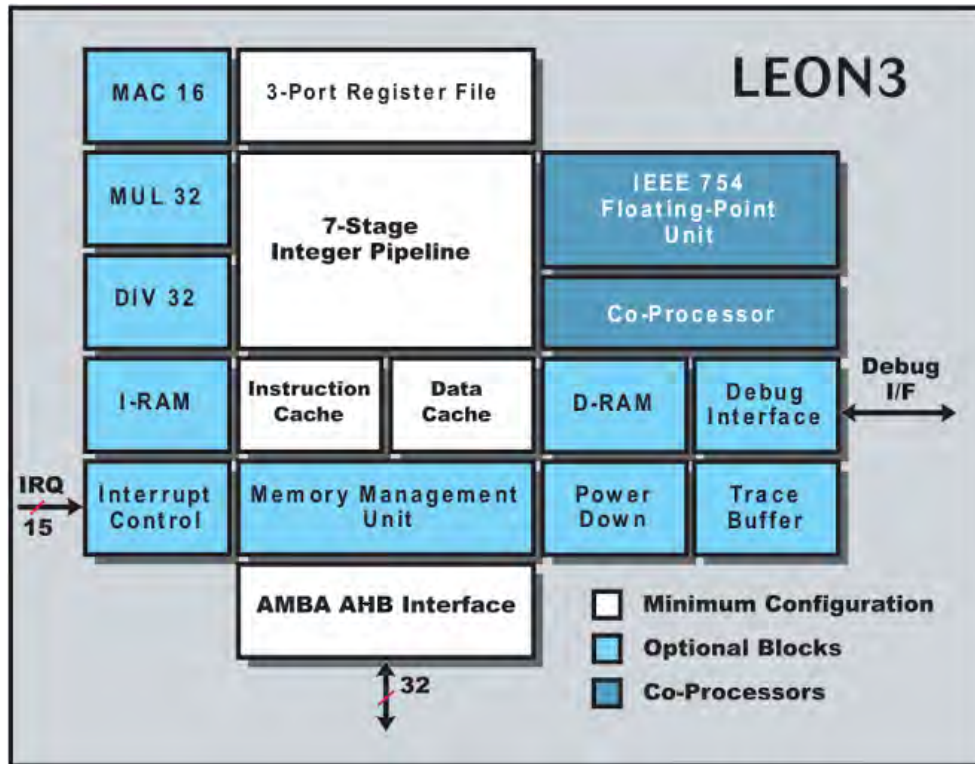


Figure 20. Diagram of a LEON3 CPU [14].

To fit four reconfigurable CPUs within the limited resources of a reasonably-priced FPGA board, only the “Minimum Configuration” blocks shown in white in addition to the “Debug Interface” should be implemented.

The LEON3 is open-source and can be modified and run on any FPGA. This is advantageous because it offers flexibility in the choice of future hardware. The LEON processor may also offer improved performance over the MicroBlaze processor.

According to a Master’s Thesis study of the subject done by Mattsson and Christensson in 2004, the LEON processor has better performance than the MicroBlaze processor [4].

Cobham Gaisler Components

This section describes the Cobham Gaisler supplied tools which are available with the LEON processor and which can all be implemented in a Linux environment to assist in development of the LEON3. Unfortunately, the USB drivers to connect to the board are very difficult to install properly for Linux. Therefore, the GRMON2 debug tool was used on Windows 7 instead.

GRLIB Processor Files are pre-defined files that can be used to create a LEON processor that can run on any of the supported Development Boards. They contain the entire source VHDL code needed to compile a full LEON processor, including all code for the memory modules. Support for numerous boards, including both Xilinx and Altera, is included. The “xconfig” graphical configuration tool is also available as a way to easily modify the design of the processor, and to turn different components on or off.

The BCC cross-compiler can be downloaded from the Gaisler website in order to compile C code to run on the LEON processor. Based on the standard Linux GCC compiler, BCC is necessary in order to compile code that can be run properly on LEON. Writing application software and compiling it with BCC Compiler is straightforward; for this project both a simple “Hello World” program and a simple “Counter” program were written. LEON is also capable of running a full-fledged Linux operating system, a sample of which can also be downloaded from the Gaisler website.

The TSIM ERC32/LEON Simulator is a simulator made especially for LEON. It was used to verify correct operation of the “Hello World” program on an unmodified

LEON processor. However, it was unclear how to run simulations based on modified LEON code, and this software was not used to test the final design.

Finally, GRMON2 is a non-intrusive debugging monitor provided by Cobham Gaisler that allows for observation of LEON processor resources. It also allows read and write access to all registers and memory.

If the professional version of GRMON2 is purchased, error injection to simulate radiation strikes is also supported. According to the GRMON User's Manual, "An injected error flips a randomly chosen memory bit in a [sic] one of the memory blocks, effectively emulating an SEU" [13]. If this could be implemented with a 4-core LEON processor, a good proof-of-concept could be verified.

LEON Implementation Instructions

The following paragraphs outline the basic process of getting the LEON3 software functioning on an AC701 development board with an Artix-7 processor. Other boards can be used as well; LEON3 comes preconfigured with basic implementations for numerous board choices. See Appendix A for more detailed instructions on implementing a LEON3 system, as well as for links to websites for downloading the various required files.

The two main things necessary are the LEON3 "gplib" software suite, downloadable directly from Aeroflex Gaisler's website, and, if development on a Xilinx Series 7 board is desired, Xilinx Vivado Design Suite. There are other software packages that can be downloaded from Gaisler as well, including a LEON-specific compiler, the

TSIM simulator, and the GRMON debug tool. These tools were described in more detail in the previous section.

Although in theory Cobham Gaisler's "GRTTools" software can be used to develop LEON in a Windows environment, in practice, synthesizing the LEON requires Linux. This project used SUSE Linux Enterprise Desktop version 12, although Red Hat and Ubuntu also work. Installing Linux and setting up the Xilinx toolchain therein are described in detail in Appendix A. Unfortunately, although Linux and the required Xilinx software were successfully installed, the drivers were not able to talk to the FPGA from the Linux computer. Therefore, actual downloading of the bitstream and communication with the board using GRMON was done from a computer running Windows 7.

The specific steps required to build and synthesize the LEON3 processor can be confusing since instructions vary for different FPGAs. Fortunately, the Artix-7 board uses Vivado Design Suite, which enables the entire synthesis, implementation and bitstream generation process to be run from the Vivado GUI, obviating the need for complex command line instructions. Simply add the appropriate paths for Vivado and ISE and then run "make vivado-launch" in the terminal to open Vivado, and work from there. For more detailed instructions, including instructions for making the LEON CPUs reconfigurable and for implementing Triple Modular Redundancy, see Appendix A. Instructions relating to implementing a similar design on a Virtex-6 FPGA may be found in Appendix B.

EXPERIMENTAL RESULTS

Four different configurations of the LEON processor were implemented. They are described below, in order of increasing complexity. For the first three designs, no changes are required to the VHDL code downloaded as part of the GRLIB library from Cobham Gaisler. All that is needed is to choose the appropriate option for the number of CPUs in the Gaisler “xconfig” GUI, as well as to place and make reconfigurable the CPUs in the Vivado GUI when appropriate. To emulate conditions on the actual hardware stack, in all designs the external memory controller is turned off and 128kB of on-chip SRAM is used instead, starting at memory location 0x40000000. All four designs were implemented on an Artix-7 XC7A200T-2FBG676C FPGA at a stepped-down clock speed of 100MHz.

Single Core LEON

This is the simplest instantiation of the LEON3 processor, requiring no special modifications. A block diagram of this configuration is shown in Figure 21.

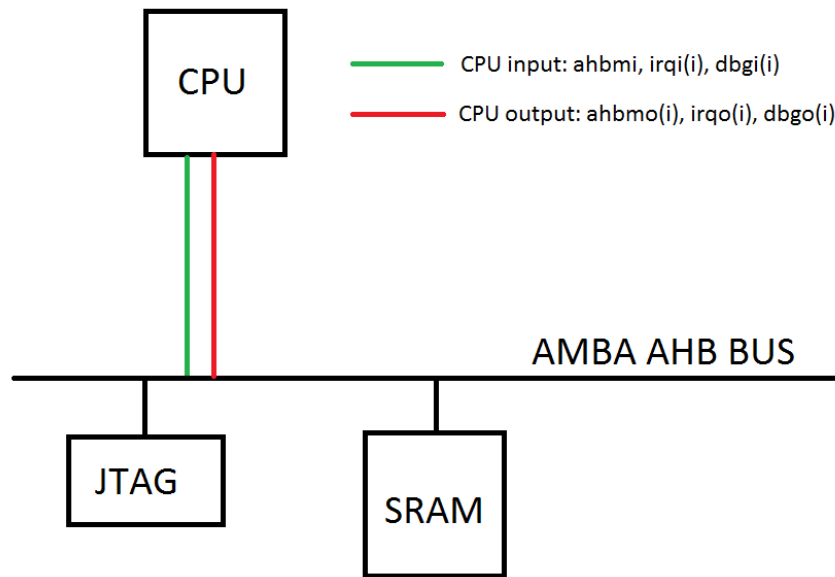


Figure 21. Block diagram of a single core LEON processor.

The resource usage on the Artix-7 FPGA is shown in Figure 22 below:

Utilization - Post-Implementation			
Resource	Utilization	Available	Utilization %
FF	2964	267600	1.11
LUT	6714	133800	5.02
Memory LUT	11	46200	0.02
I/O	16	400	4.00
BRAM	76	365	20.82
BUFG	2	32	6.25
PLL	1	10	10.00

Figure 22. Resource usage of a single core LEON processor on Artix-7 FPGA.

The implemented floorplan is shown in Figure 23.

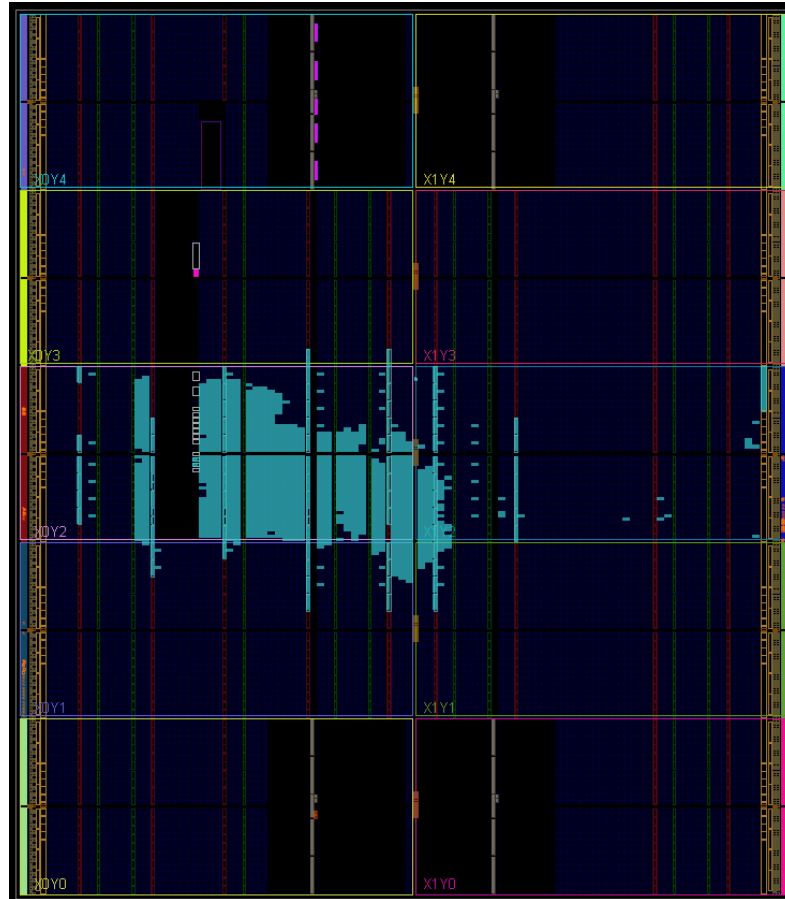


Figure 23. Implemented single core LEON processor on Artix-7 FPGA.

As shown above, the single core design without partial reconfiguration easily fits onto the FPGA and uses a small amount of the available resources. Note that the implemented design clusters near the X0Y2 clock region, which contains the JTAG communication pins.

Four Core LEON

This design is identical to the previous one, except that there are four CPUs instead of one. A block diagram of this configuration is shown in Figure 24.

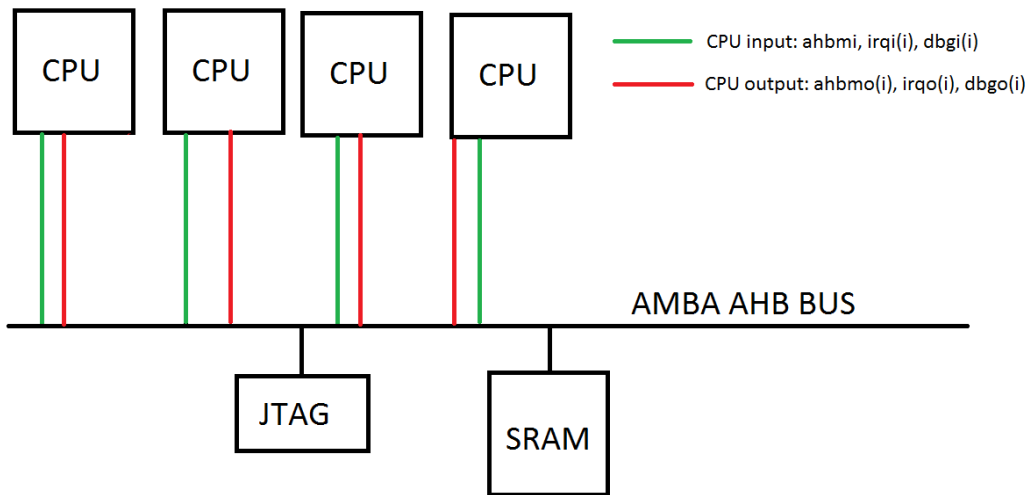


Figure 24. Block diagram of a four-core LEON processor.

The resource usage on the Artix-7 FPGA is shown in Figure 25 below:

Utilization - Post-Implementation			
Resource	Utilization	Available	Utilization %
FF	7383	267600	2.76
LUT	21071	133800	15.75
Memory LUT	41	46200	0.09
I/O	21	400	5.25
BRAM	112	365	30.68
BUFG	2	32	6.25
PLL	1	10	10.00

Figure 25. Resource usage of a four-core LEON processor on Artix-7 FPGA.

The implemented floorplan is shown in Figure 26.

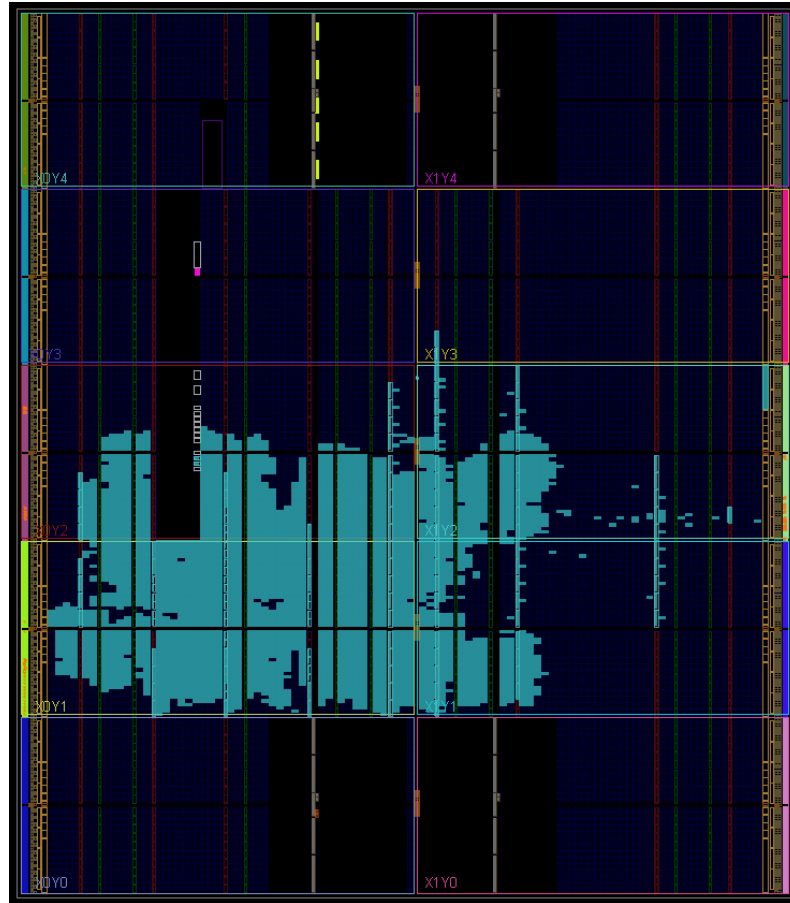


Figure 26. Implemented four-core LEON processor on Artix-7 FPGA.

As shown, the four-core LEON uses moderately more resources than the single core version, but still easily fits onto the board.

Four Core, Reconfigurable LEON

The next step involves introducing the ability to independently reconfigure each of the four CPUs. In order to fit this configuration onto the Artix-7 FPGA, the “Minimal” CPU was chosen with the Floating Point Unit option excluded, but the Debug Support Unit included (for ease of comparison, this configuration was used in all four versions). This configuration includes separate 8kB instruction and data caches in each CPU. Each of the four CPUs was assigned to two clock regions, with one CPU placed in each of the four corners of the chip. A block diagram of this configuration is shown in Figure 27 below:

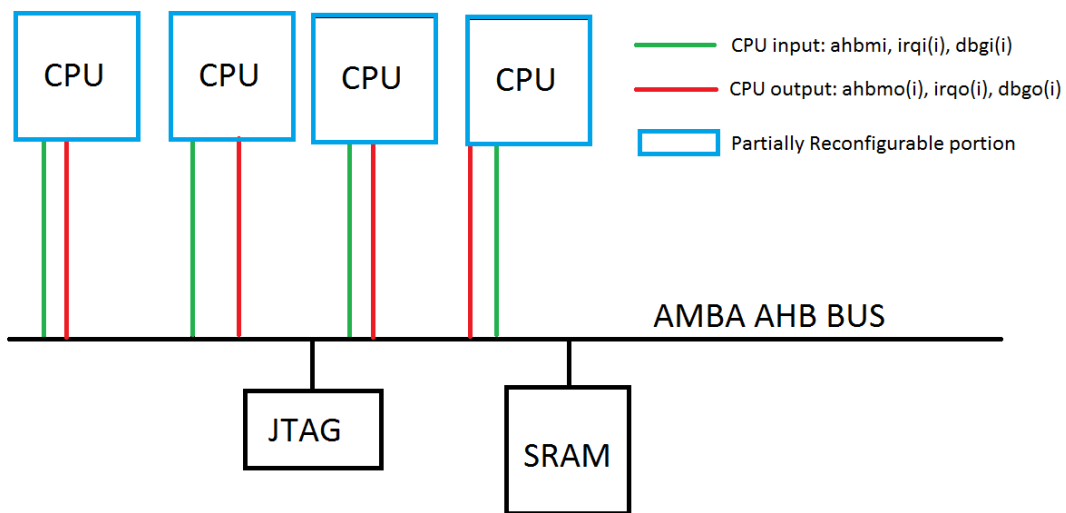


Figure 27. Block diagram of a four-core, partially reconfigurable LEON processor.

The resource usage on the Artix-7 FPGA is shown in Figure 28 below.

Utilization - Post-Implementation			
Resource	Utilization	Available	Utilization %
FF	7485	267600	2.80
LUT	55180	133800	41.24
Memory LUT	29	46200	0.06
I/O	21	400	5.25
BRAM	112	365	30.68
BUFG	2	32	6.25
PLL	1	10	10.00

Figure 28. Resource usage of a four-core, partially reconfigurable LEON processor on Artix-7 FPGA.

The implemented floorplan is shown in Figure 29 on the following page.

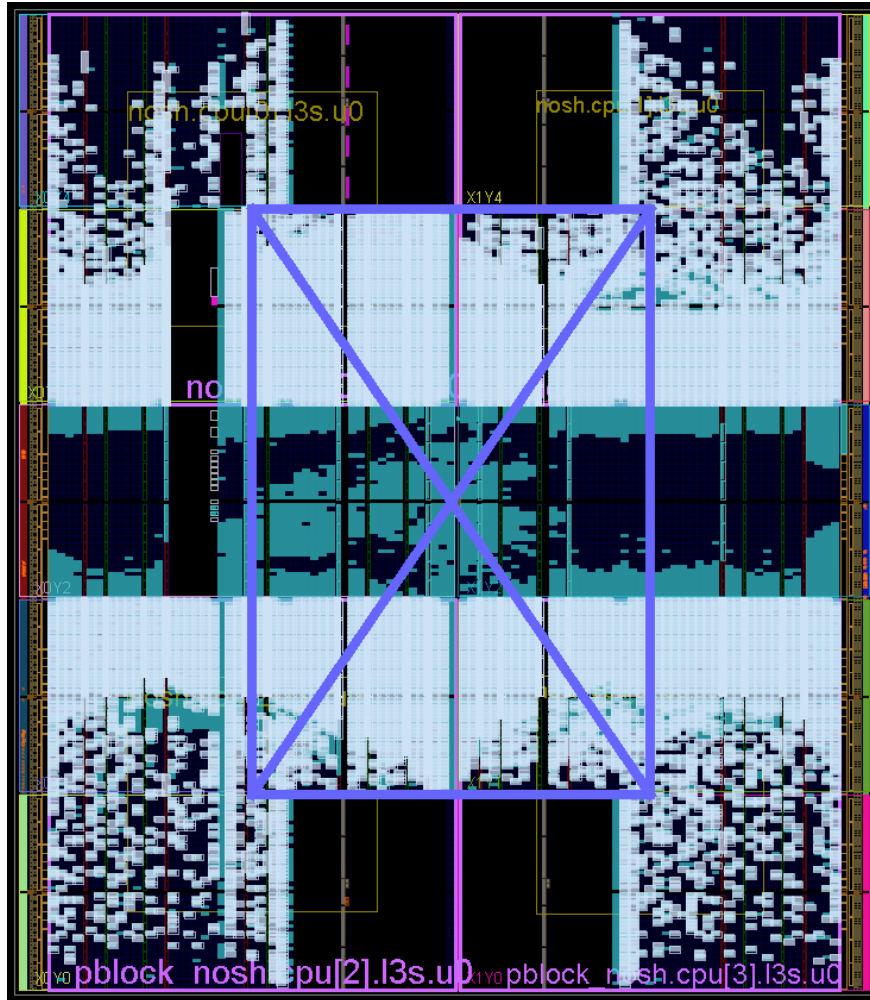


Figure 29. Implemented four-core, partially reconfigurable LEON processor on Artix-7 FPGA.

As can be seen in Figures 28 and 29, adding partial reconfiguration to the cores greatly increases the Look Up Table (LUT) usage on the board and makes the floorplan quite crowded. Each core, occupying two clock regions on one of the corners of the board, contains a complete CPU. The purple “X” pattern delineates the bus communication between the four cores and the static region. The blue resources in the middle contain the static portion of the design, including RAM, AHB bus, and JTAG communications. In

this design, each CPU is independently reconfigurable, but all four receive instructions from the same static non-reconfigurable on-board SRAM. Partial reconfigurability also causes implementation to take far longer. A four core LEON with PR takes about 20 minutes to implement, whereas the partially reconfigurable version shown in Figure 29 takes around 6-8 hours.

Four Core, Reconfigurable LEON with Triple Modular Redundancy

Next, the VHDL code was modified to implement triple modular redundancy. Since four entire LEON processors cannot fit on an Artix-7 FPGA, one processor with four CPUs was used instead. Four CPUs were created, all of which are fed the same inputs and run the same lines of code in parallel. The outputs are then fed into a voter that detects and recovers from a fault in any one of the CPUs. A block diagram of this design is shown in Figure 30 below:

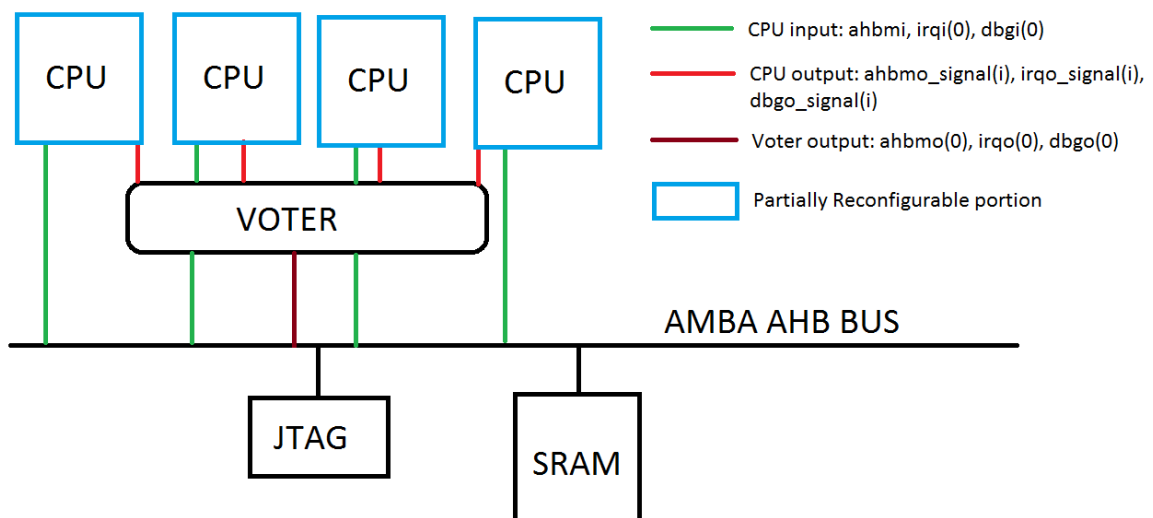


Figure 30. Diagram of modified LEON3 with voter and TMR+spare architecture.

In the original design downloaded from the Gaisler website, the four cores act as a traditional four-core processor, and each instruction is only executed by one core. With the TMR code, however, all four cores run the same instructions in parallel. Additional logic is then used to compare the outputs from the three active cores and, if necessary, bring the spare online and set flags to indicate any errors. The AHB bus input to each CPU was tied to a pushbutton on the development board, so that an error could be manually introduced into any CPU during operation, as desired.

It is worth noting that, as shown in the figure above, the voter votes on all outputs of the CPUs on each clock cycle. The counter program used in this project uses a *printf* statement to send the output to the terminal after each operation. However, if such a statement is not used, and if the caches are of sufficient size to hold all instructions, the program could potentially run for a significant period of time with an error before the error is detected. Separate 8kB instruction and data caches were included in this design; however, for the aforementioned reason it may be desired to remove caches from future designs. This would force an instruction retrieval and data output for each instruction, guaranteeing regular voter functioning. Alternatively, the CPU code could be modified to allow voter access to the internal registers and caches. However, this would involve modification to several VHDL files, something which was not done for this project.

The usage of resources on the Artix-7 with the TMR is shown on the following page in Figure 31.

Utilization - Post-Implementation			
Resource	Utilization	Available	Utilization %
FF	7438	267600	2.78
LUT	56373	133800	42.13
Memory LUT	41	46200	0.09
I/O	22	400	5.50
BRAM	112	365	30.68
BUFG	2	32	6.25
PLL	1	10	10.00

Figure 31. Resource usage of a four-core, partially reconfigurable LEON processor with triple modular redundancy on Artix-7 FPGA.

As can be seen, adding TMR increases LUT usage by a small amount but does not otherwise significantly change the amount of resources used. The implemented floorplan for this design is shown in Figure 32.

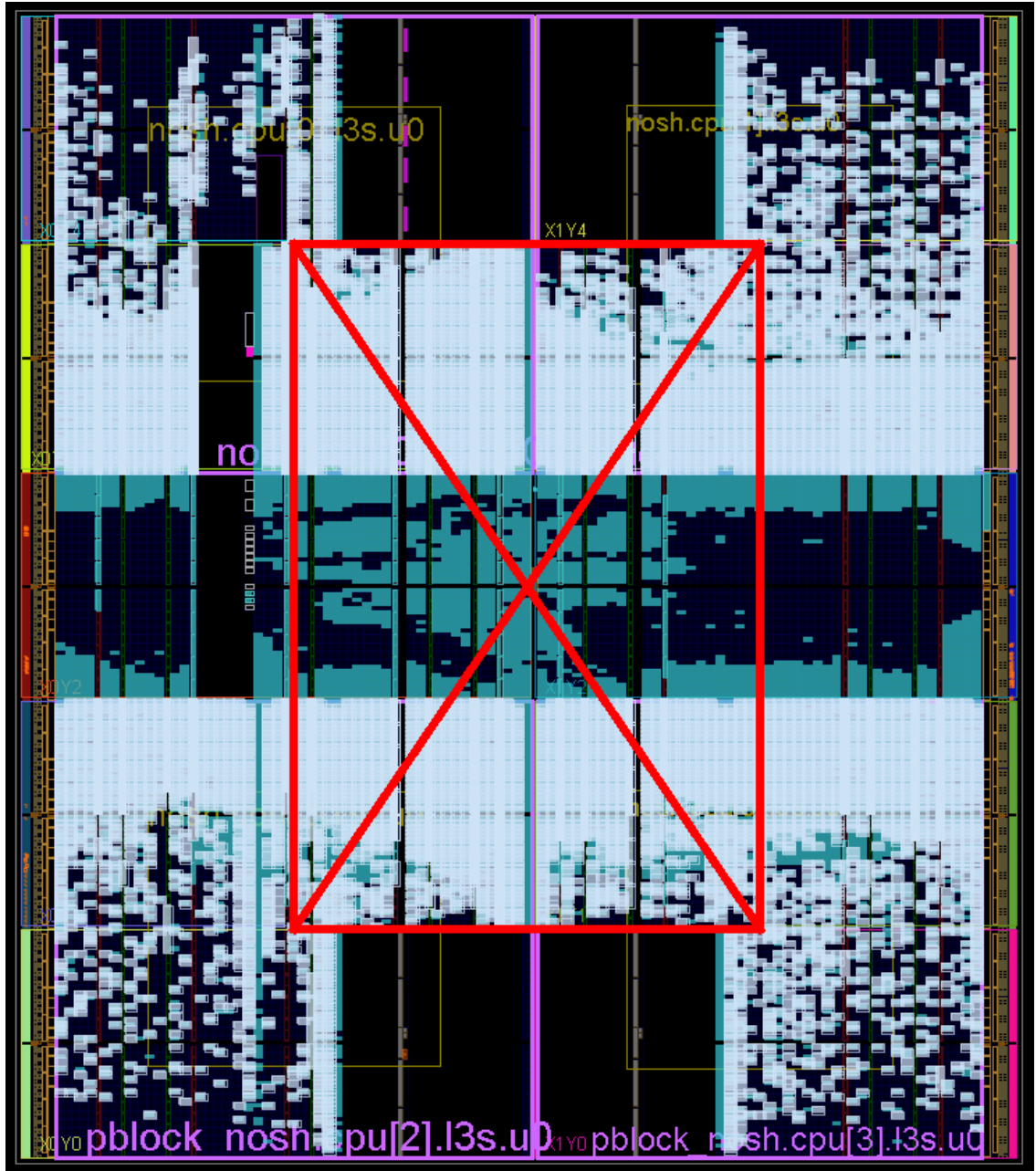


Figure 32. Implemented four-core, reconfigurable LEON with TMR on Artix-7 board.

The floorplan for this design looks substantially the same as the four-core, reconfigurable design without TMR. The resource usage of a single reconfigurable CPU, taking up two clock regions, is given in Figure 33.

Site Type	Available	Required	% Util
LUT as Logic	25600	9997	39.05
LUT as Memory	8800	10	0.11
SLICEL	4200	1974	47.00
SLICEM	2200	1197	54.41
Slice Registers	51200	1431	2.79
F7 Muxes	12800	82	0.64
F8 Muxes	6400	3	0.05
Block RAM Tile	60	12	20.00
RAMB36/FIFO	60	8	13.33
RAMB18	120	8	6.67
DSPs	120	0	0.00

Figure 33. Resource usage of a single reconfigurable CPU occupying two clock regions on Artix-7 FPGA.

The voter compares the outputs from the different CPUs and sends the correct one to the bus. The voter can detect and correct one error in one of the CPUs, but loses fault tolerance after this since the spare cannot yet be brought online to replace the faulted CPU. Figure 34 on the following page shows LEON running a counter program with TMR, in which all four cores are running in parallel.

```

grmon2> lo counter.exe
40000000 .text          47.8kB / 47.8kB  [=====>] 100%
4000BF30 .data          2.7kB / 2.7kB  [=====>] 100%
Total size: 50.52kB (592.02kbit/s)
Entry point 0x40000000
Image C:/opt/counter.exe loaded

grmon2> run
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
100, 101, 102, 103, 104, 105, 106, 107, 108, 109,
110, 111, 112, 113, 114, 115, 116, 117, 118, 119,
120, 121, 122, 123, 124, 125, 126, 127, 128, 129,
130, 131, 132, 133, 134, 135, 136, 137, 138, 139,
140, 141, 142, 143, 144, 145, 146, 147, 148, 149,
150, 151, 152, 153, 154, 155, 156, 157, 158, 159,
160, 161, 162, 163, 164, 165, 166, 167, 168, 169,
170, 171, 172, 173, 174, 175, 176, 177, 178, 179,
180, 181, 182, 183, 184, 185, 186, 187, 188, 189,
190, 191, 192, 193, 194, 195, 196, 197, 198, 199,
200, 201, 202, 203, 204, 205, 206, 207, 208, 209,
210, 211, 212, 213, 214, 215, 216, 217, 218, 219,
220, 221, 222, 223, 224, 225, 226, 227, 228, 229,
230, 231, 232, 233, 234, 235, 236, 237, 238, 239,
240, 241, 242, 243, 244, 245, 246, 247, 248, 249,
250, 251, 252, 253, 254, 255, 256, 257, 258, 259,
260, 261, 262, 263

CPU 0: Interrupted!
0x4000aa18: 02bffffe be 0x4000aa10 <outbyte+20>
CPU 1: Interrupted!
0x4000aa18: 02bffffe be 0x4000aa10 <outbyte+20>
CPU 2: Interrupted!
0x4000aa18: 02bffffe be 0x4000aa10 <outbyte+20>
CPU 3: Interrupted!
0x4000aa18: 02bffffe be 0x4000aa10 <outbyte+20>

```

Figure 34. Upon interruption, all four cores of LEON are shown to be executing the same instruction.

Assuming a functioning system for substituting the spare CPU, the system could detect and recover from two faults but would lose fault tolerance thereafter, since logic to resynch a CPU with the other three processors after partial reconfiguration is quite complicated. One of the CPUs can be reconfigured without interfering with the processes running on the others, but will be out of synch thereafter and therefore not able to contribute to future fault tolerance. In practice, it may be simpler to simply reset all four processors after one is reconfigured, and start running the code again from the beginning.

For the code used to achieve the triple modular redundancy described above, see Appendix C.

Future Four Tile, Reconfigurable Design

The next step, which was not implemented in this project, is a four tile reconfigurable system with TMR, or four entire LEON processors, each of which can be independently reconfigured. This design will not fit onto an Artix-7 FPGA, but should be able to fit onto a Virtex-7 FPGA. Alternatively, four Artix-7 FPGAs could be used, with one processor placed on each chip. Given the reasonable cost of an Artix-7 and the high cost of a Virtex-7 at present, this is expected to be a more feasible option in the coming years. However, it would necessitate a significant redesign of the FPGA board, since space for three new FPGAs would need to be found on the circuit board. While exact numbers for resource usage of such a four-tile design are not available, it is estimated that it would use about 160,000 LUT, or about 1.2x as much resources as are theoretically available on an Artix-7 200T FPGA. In order make a partially reconfigurable implementation feasible, resource usage on the order of 50% is desired. This implies FPGA resources of around 320,000 LUT, which translates to about 80,000 slices. Such resources are available from a low-end Virtex-7.

FPGA Resource Usage and Configuration Timing

The resource usage and time required to write bitstreams to the Artix-7 FPGA are given in Table 2 on the following page. The bitstream times were measured for both the

full bitstream as well as for a partial bitstream which only reconfigures a single CPU.

Estimated numbers are also given for a four-tile system.

System	Description	Resource Usage		Full config time	PR time
		LUT	%		
System 1	Single core	6,714	5.02	11s	N/A
System 2	Four core	21,071	15.75	11s	N/A
System 3	Four core with PR	55,180	41.24	11s	2.5s
System 4	Four core with PR & TMR	56,373	42.13	11s	2.5s
System 5	Four tile with PR & TMR	160,000*	120*	32s*	7.3*

Table 2. A listing of the different systems implemented in this project and their various characteristics (*estimated).

Resource usage numbers are given in both absolute terms and as a percentage of the resources available on an Artix-7 200T (215,360 logic cells and 33,650 Configurable Logic Block slices, each with four LUTs). A Virtex-7 585T, on the other hand, contains 582,720 logic cells and 91,050 Configurable Logic Block slices, each with four LUTs. It is believed, therefore, that a four-tile design should fit onto a Virtex-7 585T at about 50% resource usage [19].

Bitstream configuration times were measured with a stopwatch and are accurate to within $\frac{1}{4}$ second. Partial reconfiguration times are about two and a half seconds and therefore should not present a large problem for a space computer. In most outer space environments, radiation strikes are rare enough that a time delay of a few seconds to reconfigure a corrupted CPU should not present a large risk. Configuration times for the four-tile system are extrapolated based on the resources on a Virtex-7 585T FPGA.

Commentary on Partial Reconfiguration and Resource Usage

Implementation of partial reconfiguration in this project revealed a few practical limitations of FPGA resource usage in partially reconfigurable designs. In a traditional FPGA design not incorporating partial reconfiguration, 85-90% of resources can typically be utilized. In a partially reconfigurable design, however, such high resource usage cannot be practically placed and routed. In a partially reconfigurable region, a resource usage of around 60% is a reasonable goal. Higher resource usage may be possible with careful planning but routing becomes very difficult, even if the design can be satisfactorily placed. This is the reason that, although all resource utilization categories are well under 25% in the single-core LEON design as shown in Figure 22, it is not possible to fit four such designs on the Artix-7. The additional routing required by the partial reconfiguration drastically diminishes the design capacity of the chip.

FUTURE WORK

Several paths can be pursued in the future to further the objectives of this project. First and foremost, the design of the TMR voter needs to be modified so that the output from the spare CPU can be used in the event of a fault in one of the three main CPUs. This will allow the system to detect and recover from two faults rather than just one. Furthermore, a method should be put in place to synch the processors after reconfiguration so that operation can continue indefinitely. Also, the implemented LEON3 on the Artix-7 board should be connected to the Spartan-6 control board, so that when error flags are set, the Spartan board can reconfigure the appropriate CPU(s) on the Artix-7 board. All of this can be done with the current Artemis hardware setup.

A further step, which would require a new and larger main FPGA board, is the instantiation of four entire reconfigurable LEON processors, rather than only four reconfigurable CPUs. However, this would require a significantly larger chip, around three times larger than the Artix-7 200T chip currently in use. The only Xilinx chip which meets this requirement is a Virtex-7. The Virtex-7 585T is the recommended chip for this purpose, although it unfortunately costs about twenty times as much as the Artix-7 200T. A Virtex-7 485T may be a reasonable choice as well; it is somewhat smaller than the 585T but is available on the VC707 development board, and a pre-made design for this FPGA is included in the GRLIB library, which would make development easier. Although both feasible Virtex-7 options are very expensive, in a few years their prices may decrease enough to justify their use in the Artemis stack, allowing the goal of four entire reconfigurable LEON processors on one FPGA to take shape.

Alternately, as mentioned earlier, four separate Artix-7 FPGAs could be used, with one complete LEON processor on each FPGA. This would bring down the cost to a reasonable level at the expense of requiring significant redesign of the FPGA board. This would, however, make an excellent project for a future researcher.

If attempting to implement a four-tile design with partial reconfiguration on a single FPGA, it should be kept in mind that the JTAG communication logic cannot be reconfigured. Furthermore, all clock generators and reset signals, as presently set up, require access to specific hardware pins and must be removed from any partially reconfigurable portion of the design and placed in the static region.

Other future work could also revolve around testing the LEON3 with additional software from Cobham Gaisler. If the GRMON2 Pro software is purchased, errors could be injected into the TMR design to test the results. The GRMON2 software randomly injects errors into a memory block during operation, which more closely resembles an actual Single Event Upset than the pushbutton-operated error injection currently in use [13]. This would enable more realistic testing of the system's ability to detect and recover from radiation-induced faults. Some special GRMON code could also potentially be written to inject errors into all three active cores, test the results, and display the three different results after running a testbench code.

REFERENCES CITED

- [1] Holmes-Siedle, Andrew and Adams, Len. *Handbook of Radiation Effects*. 2nd ed. Oxford University Press, USA, 2002. Print.
- [2] Blaylock, Eva. "Novel Technology Takes Great Step Forward with Success in Flight." *Wright-Patterson Air Force Base*. 23 May 2013. Web. 09 February 2015. <<http://www.wpafb.af.mil/news/story.asp?id=123349827>>.
- [3] "LEON: The Space Chip that Europe Built." *ESA*. 7 January 2013. Web. 09 February 2015. <http://www.esa.int/Our_Activities/Space_Engineering_Technology/LEON_the_space_chip_that_Europe_built>.
- [4] Mattsson, Daniel and Christensson, Marcus. *Evaluation of synthesizable CPU cores*. Master's Thesis, Chalmers University of Technology. Gothenburg, 2004.
- [5] "Radiation-Hardened, Space-Grade Virtex-5QV Family Overview." *XILINX*. 12 November 2014. Web. 12 February 2015. <http://www.xilinx.com/support/documentation/data_sheets/ds192_V5QV_Device_Overview.pdf>.
- [6] Pragasam, Ravi. "Space Satellite Designs Need Flexible, Rad-Hard ICs." *COTS Journal*. RTC Group, August 2005. Web. 23 March 2015. <http://www.cotsjournalonline.com/articles/view/100383>
- [7] "Rendering of Van Allen radiation belts of Earth 2." *Wikimedia Commons*. Wikimedia Foundation, Inc. 25 September 2010. Web. 23 March 2015. http://commons.wikimedia.org/wiki/File:Rendering_of_Van_Allen_radiation_belts_of_Earth_2.jpg
- [8] Kayali, Sammy. "Space Radiation Effects on Microelectronics." *JPL Office of Safety and Mission Success*. JPL. Web. 28 March 2015. PDF. http://parts.jpl.nasa.gov/docs/Radcrs_Final.pdf
- [9] Meroli, Stefano. "Radiation damage for silicon detectors." *Welcome in My blog*. CERN. 19 December 2012. Web. 29 March 2015. http://meroli.web.cern.ch/meroli/lecture_radiation_damage_silicon_detector.html
- [10] Hogan, Justin. "Reliability Analysis of Radiation Induced Fault Mitigation Strategies in Field Programmable Gate Arrays." Dissertation. Montana State University, Bozeman. April 2014. PDF.

- [11] “Новое семейство радиационно-стойких микросхем от XILINX.” Macro Group. 23 August 2011. Web. 29 March 2015. <http://www.macrogrou.ru/news/2014/140>
- [12] “Artix-7 FPGA AC701 Evaluation Kit - Board Debug Checklist.” Xilinx Inc. 2015. Web. 18 May 2015. <http://www.xilinx.com/support/answers/54139.html>
- [13] “GRMON User’s Manual, Version 1.1.61.” Cobham. Aeroflex. 2013. Web. 2015. PDF. <http://www.gaisler.com/doc/grmon.pdf>
- [14] “LEON3 32-bit processor core.” Realtime Embedded AB. 31 January 2012. Web. 22 June 2015. <http://www.rte.se/blog/blogg-modesty-corex/leon3-32-bit-processor-core/1.5>
- [15] “LEON3 Processor.” Cobham. Aeroflex. 2015. Web. 9 July 2015. <http://www.gaisler.com/index.php/products/processors/leon3>
- [16] “The Design and Simulation of the Shield Reduce Ionizing Radiation Effects on Electronic Circuits in Satellites.” *Scientific & Academic Publishing*. Scientific & Academic Publishing Co. 2011. Web. 13 July 2015. <http://article.sapub.org/10.5923.j.eee.20110102.16.html#Ref>
- [17] Harkness, Sam. “Experiment Platform to Facilitate Flight Testing of Fault Tolerant Reconfigurable Computer Systems.” Thesis. Montana State University, Bozeman. May 2015. PDF.
- [18] A. Keys, J. H. Adams, J. D. Cressler, M. C. Patrick, M. A. Johnson, R. C. Darty. Radiation hardened electronics for space environments (rhese) project overview. *Sixth International Planetary Probe Workshop*, 2008.
- [19] “7 Series FPGAs Overview.” Xilinx Inc. 27 May 2015. Web. 18 July 2015. PDF. http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf
- [20] “MSU computer reaches space aboard research rocket.” *MSU News*. Montana State University. 23 October 2014. Web. 18 July 2015. <http://www.montana.edu/news/15193/msu-computer-reaches-space-aboard-research-rocket>
- [21] “Dragon’s ‘Radiation-Tolerant’ Design.” *Aviation Week Network*. Penton. 2015. Web. 19 July 2015. <http://aviationweek.com/blog/dragons-radiation-tolerant-design>

APPENDICES

APPENDIX A

DETAILED INSTRUCTIONS FOR LEON

Installing LEON

Here I am documenting the process to get the LEON3 processor up and running on an Artix-7 FPGA board. First of all, the processor and related implementation code can be downloaded as “[gplib-gpl-1.3.7-b4144.tar.gz](http://www.gaisler.com/index.php/downloads/leongrplib)” from <http://www.gaisler.com/index.php/downloads/leongrplib>. Manuals are also available on this same page. Other items are available for download from the same website under the “Downloads” tab.

LEON and the Linux Toolchain

Setting up the toolchain necessary to modify the LEON processor code and download it onto the target board can be a tedious and time-consuming process if the correct environment is not chosen. For this project, SUSE Linux Enterprise Desktop (SLED) 12 was used to set up the project and run synthesis. Then, either Windows 7 or SUSE 12 can be used to run implementation and generate a bitstream.

Linux is not absolutely required for working with the LEON3 code; the design can also be implemented from within a UNIX shell on Windows and using GRTools. However, Linux is LEON’s native environment and makes for the easiest going. Therefore, it is recommended that Linux be installed in order to synthesize the design. The best results in this project were achieved using the SUSE 12 Desktop. However, this version of Linux does not unfortunately support downloading the code onto Xilinx boards, so the completed bitstream was manually transferred back to Windows and

downloaded to the AC701 development board from there using Xilinx iMPACT software.

Getting the LEON processor running, even with the correct operating system, can be a confusing process depending on the FPGA board targeted. Fortunately, the Vivado development environment used for the Artix-7 board is relatively straightforward to use. Various support programs can also be downloaded and integrated with the main software in order to simulate or help debug it.

Running Xilinx Software on Linux

Install Red Hat or SUSE Enterprise for best results (you can get a free trial version of either at the following websites):

Buy SUSE: from \$50 <https://www.suse.com/products/desktop/how-to-buy/shop.html>

Buy Red Hat: from \$49 <https://www.redhat.com/apps/store/desktop/>

First, I installed Linux in dual-boot mode directly onto a computer and tried to work with that. First I installed Ubuntu 14.10 for 64-bit system on the lab computer ece-hsddl1. Next I installed SUSE Linux Enterprise Desktop version 12 on a different lab computer, which I subsequently used for all important tasks.

Then install the proper version of Xilinx for Linux. Supposedly you are supposed to have both ISE and Vivado installed. I went to

<http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html> and downloaded “ISE Design Suite 14.7” via the file “[Full Installer for](#)

[Linux](#) (TAR/GZIP - 6.09 GB)". Also download Viado (v. 2014 or later) from the same website. You can also download the same thing in 3 separate files if desired. Then I reformatted a flash drive to exFAT so it could hold the 6.09 GB install file for ISE and transferred it to the Linux computer. I then installed it in the /opt directory under /opt/Xilinx/14.7/ISE_DS.

Concerning Xilinx Vivado software, follow the instructions in the AC701 design README file, and make sure that links to both Vivado and ISE directories are in your path. Enter "make xconfig" to setup the design parameters you want. Then, either enter "make vivado-launch" to do the design in the GUI, or "make vivado" to synthesize everything in the command line, all at once. You need to have the correct version of Vivado installed, or else the memory controller will not compile properly. Check the README file for the recommended version of Vivado. For the Leon files I am using, I had to install Vivado 2013.3 to get the necessary MIG compiler, which is version 2.0. But for general purpose, I used Vivado 2014.4.

To get a Xilinx license for a Linux machine, you need the MAC to install a node-locked license. Find this by looking at the "HWaddr" field when you type "/sbin/ifconfig" into the terminal. For the machine I used, the MAC number is 001e4fdf4880.

Once you have the license file (should be called "Xilinx.lic"), open the license manager in Xilinx on Linux: go to directory /opt/Xilinx/14.7/ISE_DS/common/bin/linux on the command line, and run ./xlicm . Then just load the proper license file for your

computer in the GUI that should appear. After this is done, Xilinx ISE (which includes CORE Generator) should be working.

Next step is to set up some variables so that Core Generator can work properly. Open up a terminal window and navigate to the folder containing your project files (for me it was /Desktop/Leon/glib-gpl-1.3.7-b4144/designs/leon3-xilinx-ml605). Once there, run the command “env” (that step may perhaps not be strictly necessary). Then, run the command “source /opt/Xilinx/14.7/ISE_DS/settings64.sh”. This sets up some variables so that ISE and CORE Generator will function when called from the ac701 project folder. Note that this step will need to be run every time you restart the computer so that your environment variables get set properly for that session.

At this point, you can type “ise” and/or “coregen” into the terminal to open these programs and make sure they are installed properly. Then close them because we still need to do more command line stuff to compile the code properly. Inside of a project folder, you can also do “make distclean” if you feel like reverting the contents back to their original state (hopefully).

Using iMPACT on Windows

I gave up on getting the USB drivers to work in Linux to talk to the FPGA board, and instead decided to do the needed programming on the Windows machine where they are already set up. So the bitfile to download is called leon3mp.bit. This is the file to program the FPGA with. So I transferred the file to the Windows computer. And here are the instructions to put it on the FPGA:

First, start iMPACT. I typed “impact” into the program search and clicked on the result. A terminal window appeared and then in about 10 seconds, the Impact program opened. If the AC701 board is plugged into the computer it should show up on the screen in Impact as “xc7a200t”. Right-click on the device icon and select “Assign new configuration file”. Then select “leon3mp.bit”. Then, right-click again and select “Program”. If all went well, the code should now be written onto the AC701 board!

GRTools on Windows 7

Basically, the previous Linux approaches work, except that as noted, I have so far not been able to get the drivers for actually talking to the Xilinx board working on any Linux system. In addition, it will be easier in the future if somebody can just run everything on Windows rather than installing a whole new operating system. With these considerations in mind, I hereby write down the steps in my path toward getting GRTools, the full Leon toolchain for Windows, working on Windows 7. Note that I never got this to work in the end, but my efforts are recorded here nonetheless.

So here I will detail the path toward getting GRTools to work. GRTools is basically a package that can run all of the tools necessary for Leon: compile code, synthesize a design, simulate it, download it onto a board, and debug it, you name it, GRTools should do it all. First off, download GRTools from the Gaisler website at <http://www.gaisler.com/index.php/downloads/grtools>. I then ran the installer and installed all of the tools in C:\opt. Make the opt folder yourself if necessary, but this is

where everything should be installed, since that is where they would be installed on a Linux machine.

The install Tcltk since this is apparently not included in the GRTools download. This will be needed to display the GUI associated with xconfig. The appropriate package can be downloaded at <http://sourceforge.net/projects/wintcltk/files/WinTclTk/8.4.19/>. I installed it in the /opt folder along with everything else. Then, in order to make TclTk work out right later, go to the folder C:\opt\WinTclTk\bin and rename “wish85.exe” to “wish84.exe”.

Then, to open the command line so that we can run all of these programs Linux-style, go to the Windows search-thingy in the bottom left and type in “cmd”. Right-click and open as Administrator. Then, navigate to C:\opt. GRTools added some environment variables to PATH but we will need to add some more to make everything work. Normally, to add a variable to path in Windows command line, type “setx path %path%;c:\opt\path1\path2”. However, this will probably not work from command line because of a 1024 character limit.

To bypass this, we need to edit the PATH variable directly. To do this, go to Start->Computer->System Properties->Advanced System Settings->Advanced->Environment Variables. From here click on “path” and then “edit” and add on the variables that you want (you will probably need Administrator privileges to do this). Here are the variables that I added to the path:

;C:\opt\sparc-elf-4.4.2-mingw\bin;C:\msys\1.0\bin;C:\Users\z64q326\Desktop\School Stuff\Research\LEON\glibc-gpl-1.3.7-

b4144\bin;C:\opt\WinTclTk\bin;C:\Xilinx\14.4\ISE_DS\ISE\bin\nt64

You may need to modify the paths slightly depending on where your files are and what program versions you are using. Once the paths are added, you can run any of the programs by typing in the appropriate names. You can also type “start” before the program name in order to make it open in a new window. To exit a program, press Ctrl+C.

Back to the command line. Enter “C:\Xilinx\14.4\ISE_DS\ISE\.settings64.bat”.

Then navigate to the design folder where the files are, e.g. \glibc-gpl-1.3.7-

b4144\designs\leon3-xilinx-ml605. Then enter “make xconfig”. If TclTk is installed properly this should work. Then “make mig39”, then “make pcie” (those instructions are for the Virtex-6 board; I didn’t try to do those things for Artix-7).

Next, enter “make -n install-secureip > foo”. Then “notepad foo” to open the file in Notepad. Once in Notepad, remove all \ backslashes at the end of lines, and replace all the remaining \ backslashes with / forward slashes (Edit->Replace). Then save and close the file and type “bash foo” in the command terminal. This will run the updated version of install-secureip. Then, enter “make planahead”.

Artix-7 Work

First of all, we ordered an AC701 evaluation board containing an Artix-7 XC7A200T-2FBG676C FPGA. The AC701 uses a Digilent JTAG chip instead of a

Xilinx Platform chip which is used on the ML605 board. Therefore, GRMON should be started with the command “grmon -digilent -u -nb” in order to connect. Luckily, the Digilent drivers automatically installed themselves for me. If this is not the case for you, I recommend looking at the GRMON instruction manual section 4.4.7. According to this section, you need “Digilent Adept System” software installed on your computer.

Now, GRMON will recognize the LEON processor on the Artix-7 board and talk to it. To compile a program into LEON-readable form, use the sparc-elf compiler. Make sure sparc-elf is in the path, as detailed in the above GRTools section. Enter into the command line “sparc-elf-gcc-3.4.4 -O2 -msoft-float hello.c -o hello.exe” to compile the program “hello.c”, for example. It can then be loaded and run on the LEON processor from within LEON. Note that the “-u” option when starting GRMON is necessary in order to be able to see text output in the terminal.

So I successfully synthesized and compiled a design (both single-core and four-core) in Vivado, took the bitstream, transported it to my Windows computer, and then downloaded it to the Artix board via iMPACT. Then, I connected to it with GRMON using instructions above, and loaded Linux on it (“lo image.ram”, then “run”), and it all works!

Next, I put together a no-MIG version. To do this, first go into the xconfig GUI and indicate “no”, you do not want to create a DDR3 memory controller. Then, go to the on-chip RAM section and set the size of AHB on-chip RAM that you want. Default is 4kB, I changed this to 128kB. Going over 512 kB seems to fail due to a lack of room on the chip. Also, make sure you set the RAM start address to “400” (short for

0x40000000), which is where programs load to and run from (usually this is where DDR starts). If you don't do this, your programs will not load to your on-chip memory and you won't be able to use it without extra effort. Finally, go into the `leon3mp.vhd` file and comment out the "ddrdummy" section since this causes errors if left to run in the no-MIG scenario. Note that without external memory you probably won't be able to run Linux, but you should still be able to run a simple counter program.

Partial Reconfiguration with Artix-7

Since the Artix-7 uses Vivado rather than ISE software, the Partial Reconfiguration flow is somewhat different. First of all, read the UG947 PR tutorial and follow the examples contained therein. Vivado makes extensive use of Tcl scripts for Partial Reconfiguration related activities. If you run into an error about "too many positional options" from copy-and-pasted instructions, retype the dashes so that they are the shorter, proper dashes.

Initially, for the four tile design, I split up the design into two VHDL files, the top one is `leon3mp.vhd` and contains the JTAG port, the clock generator, and instantiates the actual Leon processors as well as mux logic to tell if one of the processors has produced an error. The processor itself is located in the file `leon3mpsingle.vhd`. In order that this new file `leon3mpsingle.vhd` is included in the project, go into the `/vivado` folder within the project folder and open the file `leon3mp_vivado.tcl`. Add into here the line "read_vhdl – library work leon3mpsingle.vhd".

Note that in order to run Partial Reconfiguration on an Artix-7 board, you need to be running Vivado 2014 or later. But to compile the DDR controller currently supplied (as of Spring 2015), you need Vivado 2013.3. The current design doesn't use the DDR controller so I am just using Vivado 2014.4. But if needed, you can have both installed and use the appropriate version as necessary.

Partially Reconfigurable Four-Tile Design with Artix-7

In order to make the entire LEON processor reconfigurable, as opposed to just the CPUs, the following steps must be taken. First, comment out or move to the top level all "pads", that is entities named "*_pad". These cannot be reconfigured. I commented most of them out and just moved "reset_pad" and "clk_pad_ds" to the top level. Also I moved the entire "clk_gen0" to the top level as well as "ahbjtaggen0". The reason is that clock must continue running during reconfiguration, and the JTAG is connected to specific physical pins and needs to remain in the static section. Furthermore, to make your partially reconfigurable life easier you should reduce the clock speed on the design, default is 100MHz but I am currently trying out 25 MHz to see if that helps (update: didn't seem to help).

I was also getting an error about inputs not having IOBuffers. This was only a problem on inputs into the top level that were also inputs into leon3mpsngle. To remedy this, I removed all said inputs, so that no input into the top level is also an input into the leon3mpsngle.

The tiled design synthesizes and implements unfortunately only if given an overly large area, but does not generate a bitstream. When trying to do a reconfigurable design, the design is apparently not able to be efficiently routed, even when nearly the entire board is used for the reconfigurable partition. This causes a bunch of pins in the static region to be used for buffers and GND and VCC etc. Many nets will be in partial conflict for routing resources. The implementation seems to only work if default strategies are used for the implementation.

Downscaling to reconfigurable CPUs only

I'm pretty sure that putting four entire reconfigurable LEON processors on an Artix-7 board is impossible. Because of this, I downgraded the project to merely having four reconfigurable CPUs. To start out with I just have the original leon3mp.vhd file, with no modifications made to it yet. Then, go to the configuration GUI ("make xconfig" at terminal) and remove everything you don't need, including Serial Debug Link, I2C master, and round-robin arbiter, and make it a no-MIG version as described above. Then, most importantly, go to Processor and change to minimal-configuration. Then, change to custom-configuration and add back in the Debug support unit (DSU). Now the design should be small enough to fit four reconfigurable processors on the chip.

To make the CPUs reconfigurable, enter the following instructions, either in the Tcl console or straight in the xdc file, for each reconfigurable core. The instructions shown below are for the first core, core 0:

```

set_property HD.RECONFIGURABLE 1 [get_cells nosh.cpu[0].l3s.u0]
set_property SNAPPING_MODE ON [get_pblocks pblock_nosh.cpu[0].l3s.u0]
set_property RESET_AFTER_RECONFIG true [get_pblocks pblock_nosh.cpu[0].l3s.u0]

```

The HD.RECONFIGURABLE command makes the cell partially reconfigurable, the SNAPPING_MODE commands configures resource usage to make it appropriate for a reconfigurable section, and RESET_AFTER_RECONFIG sets the CPU to revert back to a known initial state after reconfiguration.

You should now have a 4-core, reconfigurable LEON working on the Artix-7 board.

Implementing TMR with Spare

The last step is to send all four processors the same instructions simultaneously and make sure they all execute in parallel. To do this, first change the first “i” in the generic map to “0”, so each processor thinks they are the first one. Then, go to the port map and change ahbmo(i), irqo(i) and dbgo(i) to ahbmo_signal(i), irqo_signal(i), and dbgo_signal(i), respectively, so that you have control of the outputs and can only send back the ones you decide. Then, change irqi(i) to irqi(0) and dbgi(i) to dbgi(0), so that each processor gets the instructions to execute that normally only the first processor would get. This will ensure that all four processors will execute the same instructions. Also, create a new variable to instantiate the four processors (I used num_procs := 4), and set CFG_NCPU to 1. Otherwise, the controller will get upset and not work because it won’t be able to see all the processors. Then, instantiate some voter and mux code to compare the ahbmo_signal, irqo_signal, and dbgo_signal parts and determine a majority

vote. Then, send this part of the output back. Sample code: `ahbmo(0) <=`
`ahbmo_signal(Nmux); irqo(0) <= irqo_signal(Nmux); dbgo(0) <= dbgo_signal(Nmux)`,
where Nmux is the part of the output decided on as correct by the majority voter. It is
important that the ahb controller only receives back ahb, irq, and dbg outputs in the (0)
position, otherwise it will know something is fishy and will not execute properly.

Basically you have to instantiate four CPUs but make the rest of the processor think that
there is only one. As mentioned, I changed the CFG_NCPU constant to 1 and created my
own constant called “num_procs” to instantiate the four CPUs. This way the LEON
thinks there is only one CPU and doesn’t cause any fuss over my redirection of the bus
signals. See Appendix C for more detailed code.

Running Linux on LEON

This would only be possible if the design uses a DDR memory controller. See
overview at
http://www.gaisler.com/index.php/products?option=com_content&task=view&id=63 and
list of relevant downloads at <http://www.gaisler.com/index.php/downloads/linux>.
Linuxbuild is apparently the main tool with which to compile Linux images for running
on LEON. There are also some pre-made images supplied on the Gaisler website. Also
an in-depth tutorial at http://jorisvr.nl/leon3_linux.html. I tried loading the pre-built
“image.ram” and running it on the LEON via GRMON, but I got an “Unknown
watchpoint hit” error. So I restarted GRMON with the “-nb” option, which means don’t
break on error traps. This time it worked!

Linux starts, and I log in as “root”. Apparently this image is running a “Buildroot” version on Linux. See <http://buildroot.uclibc.org/downloads/manual/manual.html> for an overview of Buildroot. The system also apparently uses BusyBox, a lean version of Linux commands in order to provide mostly complete functionality without using up too much space, see <http://www.busybox.net/>. Enter “busybox” into the command line to see a list of options. I also attempted to write my own customized Linux image using the Gaisler software running on SUSE 12, but ran into errors while trying to run “Linuxbuild” because I couldn’t properly install and use the “QT” software for the qmake command. Therefore this aspect of the project was put on hold indefinitely.

APPENDIX B

ADDITIONAL INSTRUCTIONS FOR LEON3 ON VIRTEX-6

ML605 Instructions

In the beginning, I tried to compile the LEON3 VHDL code on Windows rather than on its native Linux. Most of the online tutorials I found recommended compiling the LEON core by running the “make xconfig” command in a GNU hash shell such as MSYS or Cygwin. However, I downloaded both of these shells but was unable to get either of them to properly compile the code. It seemed that some libraries or add-ons were missing from the version of Cygwin I downloaded, which prevented a GUI from appearing that would have enabled me to simulate or compile the processor code. The next step I tried was just using a stand-alone GUI to work with the code. I tried both Eclipse as well as the Xilinx ISE software. I would recommend ISE since the Eclipse does not seem to offer any benefit over ISE, aside from being open-source.

One major note about using ISE: the LEON libraries come packaged in such a way that, for example, version.vhd is in the stdlib folder, which is in the grlib folder. The version.vhd file is then called as use grlib.version.all;. However, ISE does not apparently recognize this nested structure, so the file must be placed directly in the grlib folder in order to be found (the actual file need not be moved, it just needs to be placed in grlib within the project library).

The next problem was that there were a bunch of .in.vhd files that declare constants for use in the code, but ISE keeps saying that they have syntax errors and won't use them. Therefore, the constants go undeclared when attempted to be used by the main file! The solution is to manually go through and input the correct constant values for the

specific hardware that we are running. These constants are all located in in.vhd files, the correct constants to input can be found in the associated in.help files.

Advice: consult the LEON & GRLIB Configuration & Development Guide pg. 11 for recommended values with which to fill in the in.vhd files.

Note: the above steps didn't work because I was trying to do it without GRTools. See section on GRTools below.

I ran into problems with the Cygwin shell because it could not properly run "make xconfig", the first step in the process, because it could not launch the GUI from within the shell. I don't know why this is, as I downloaded all the tcl/tk files and whatnot, but at this point I gave up on this approach.

I next tried to installing VirtualBox, installing Linux in the VirtualBox, and then running ISE on there. I successfully set up VirtualBox but and was able to install SUSE Linux Enterprise Desktop 10, but was unable to properly run the LEON sequence from therein.

Compiling for ML605 on Linux

So I gave up on doing the above steps in Windows and just went over to a real Linux machine. The command "make xgrlib" launches a GUI. So, either type "make xconfig" in the terminal, or press the button on the xgrlib GUI. A GUI should pop up, allowing you to configure your desired settings for the LEON3 processor. Choose desired settings and click "save and exit". You can also run the command "make ise-launch" to launch the ISE GUI.

At this point, you can follow the instructions contained in the “readme.text” file in the ml605 design folder under the section “Simulation and synthesis for ISE-14” (NOT ISE-13). Said file includes instructions for simulating the leon3, which I never did since I kept getting an error.

Type “make mig39” at the command line. This will generate the Xilinx memory control block. Then “make pcie”. Then “make install-secureip”. Next, to build the design, type “make planahead”. If executed successfully, this command will create a bitstream file with the compiled processor called “leon3mp.bit” that is ready for download on the FPGA board.

Communicating with the ML605 via iMPACT

The default command to do this is “make ise-prog-fpga”. However, unless the proper drivers are installed this will not work. The computer program we want to use is Xilinx “iMPACT”. One way to get this open is to open ISE, go to the design tab, highlight the main file (leon3mp.vhd in this case), expand “Configure Target Device”, and select “Manage Configuration Project (iMPACT)”. The other method is to go to /opt/Xilinx/14.7/ISE_DS/ISE/bin/linux and double-click on “impact”.

GRMON for Virtex-6 on Windows 7

To talk to the Virtex-6 with GRMON, we use the Xilinx Platform USB chip embedded on the ML605 that is connected through the USB-to-JTAG port. The proper command to connect is “grmon -xilusb -u -nb”. However, this option requires libusb-

win32 driver to be installed. Instructions and download link available at <http://sourceforge.net/p/libusb-win32/wiki/Home/>. Follow the instructions for “Filter Driver Installation”.

After downloading and extracting, go to the subfolder /bin/amd64 (or as appropriate). Copy the file libusb0.dll into the C:\Windows\system32\ folder, and copy the file libusb0.sys into the C:\Windows\system32\drivers\ folder, as described in the document found at <http://www.pinguino.cc/download/doc/libusb-windows7.pdf>. Then, run the Install Wizard “install-filter-win.exe”, proceed with “Install a device filter” option and install for the Xilinx ML605 USB device. After doing this, the ML605 USB device will have two drivers: the windrvr6 (which allows Xilinx iMPACT to communicate with it) and libusb0, which GRMON uses.

If you have driver issues, one option to look into, although it did not work for me, is playtag, an alternative to GRMON that is based on Python and attempts to communicate with the LEON through the JTAG connections. Their website is <https://code.google.com/p/playtag/>

ML605 Partial Reconfiguration Notes

First, make sure to synthesize any lower-level ngc files without I/O Buffers. Try to start planahead with the option “make planahead –out_of_context”. Also, when adding your reconfigurable modules, uncheck the “Copy sources into project” box in Planahead, so that if you update your lower-level projects, these updates will also be reflected in Planahead.

I did this, but I kept getting errors that not enough resources were available, even if I gave the entire chip to the reconfigurable partition for one Leon processor. It said that I needed 8 BUFGCTRL for example, but there are 0 available. I asked online for help but was not been able to get this fixed.

APPENDIX C

QUAD-CORE LEON (TMR WITH SPARE) VHDL CODE

```
-- Following are excerpts from the leon3mp.vhd VHDL code used for the 4-core,
TMR+PR design especially concerning the voter and error insertion
--constant num_procs : integer := (4); -- number of CPUs to implement (uncomment and
place up above if it's not there already)
```

```
-- in the future may want to send these signals as outputs back to the Spartan controller
signal Nmux : integer := 0; -- this is the variable that determines which processor's
output goes to the real outputs
```

```
    -- Nmux = 0 means that CPU0 will be chosen, etc.
```

```
signal Actproc1 : integer := 0;      -- first active processor.
signal Actproc2 : integer := 1;
signal Actproc3 : integer := 2;
signal Spareproc : integer := 3; -- this is the processor being held in spare capacity,
though it still executes the program
```

```
signal Procrun : std_logic := '1';   -- variable determining whether or not the
processors should continue to run (currently not effective)
signal Procflag : std_logic := '0';   -- flag set to 1 if there is any discrepancy
between output of the three active processors
signal Procerror : std_logic := '0';  -- error set to 1 if none of the three active
processors agree on their output
signal Procfaults : std_logic_vector(num_procs-1 downto 0) := (others => '0'); -- this
signal will have each bit set to 1 when there is an error
```

```
    -- in that
```

```
processor, or be zero if said processor is healthy
```

```
-- intermediate signals for outputs coming out of the processors
signal ahbmo_signal : ahb_mst_out_vector := (others => ahbm_none);
signal irqo_signal : irq_out_vector(0 to num_procs-1);
signal dbgo_signal : l3_debug_out_vector(0 to num_procs-1);
```

```
signal ahbmo_signal0 : ahb_mst_out_vector := (others => ahbm_none);
signal irqo_signal0 : irq_out_vector(0 to num_procs-1);
signal dbgo_signal0 : l3_debug_out_vector(0 to num_procs-1);
```

```
signal ahbmi_signal : ahb_mst_in_vector := (others => ahbm_in_none);
```

```
signal ahbmi_hrdata0 : std_logic_vector(AHBDW-1 downto 0);
signal ahbmi_hrdata1 : std_logic_vector(AHBDW-1 downto 0);
signal ahbmi_hrdata2 : std_logic_vector(AHBDW-1 downto 0);
signal ahbmi_hrdata3 : std_logic_vector(AHBDW-1 downto 0);
```

```
begin
```

 -- Here is the voter and multiplexer code:-----

```

    process(clkm)
    begin
        if ((dbgo_signal(Actproc1) = dbgo_signal(Actproc2)) AND
            (dbgo_signal(Actproc1) = dbgo_signal(Actproc3)) AND (ahbmo_signal(Actproc1) =
            ahbmo_signal(Actproc2)) AND (ahbmo_signal(Actproc1) = ahbmo_signal(Actproc3))
            AND (irqo_signal(Actproc1) = irqo_signal(Actproc2)) AND (irqo_signal(Actproc1) =
            irqo_signal(Actproc3))) then
            Nmux <= Actproc1; -- use first processor if all are behaving
normally--
            Procflag <= '0';           -- don't output a flag
            Procerror <= '0';         -- no error because we still have two
processors that agree
            led(2) <= '0';
            elsif dbgo_signal(Actproc1) = dbgo_signal(Actproc2) AND
            ahbmo_signal(Actproc1) = ahbmo_signal(Actproc2) AND irqo_signal(Actproc1) =
            irqo_signal(Actproc2) then
            Nmux <= Actproc1; -- use first processor if first two are
behaving normally
            Procflag <= '1';           -- output flag because one of the
processor returned mismatched data
            Procerror <= '0';         -- no error
            -- Actproc3 <= Spareproc; -- replace faulted processor - Uncomment
these once reconfig and resynch stuff has been solved
            -- Spareproc <= Actproc3; -- turn faulted processor into spare
            Procfaults(Actproc3) <= '1'; -- signal that active processor 3 is
now faulted
            led(2) <= '1';
            elsif dbgo_signal(Actproc1) = dbgo_signal(Actproc3) AND
            ahbmo_signal(Actproc1) = ahbmo_signal(Actproc3) AND irqo_signal(Actproc1) =
            irqo_signal(Actproc3) then
            Nmux <= Actproc1; -- use first processor 1 and 3 are behaving
normally
            Procflag <= '1';           -- output flag because one of the
processor returned mismatched data
            Procerror <= '0';         -- no error
            -- Actproc2 <= Spareproc; -- replace faulted processor
            -- Spareproc <= Actproc2; -- turn faulted processor into spare
            Procfaults(Actproc2) <= '1'; -- signal that active processor 2 is
now faulted

```

```

        led(2) <= '1';
        elsif dbgo_signal(Actproc2) = dbgo_signal(Actproc3) AND
ahbmo_signal(Actproc2) = ahbmo_signal(Actproc3) AND irqo_signal(Actproc2) =
irqo_signal(Actproc3) then
        Nmux <= Actproc2; -- use second processor if 2 and 3 are
behaving normally
        Procflag <= '1';          -- output flag because one of the
processors returned mismatched data
        Procerror <= '0';        -- no error
        -- Actproc1 <= Spareproc; -- replace faulted processor
        -- Spareproc <= Actproc1; -- turn faulted processor into spare
        Procfaults(Actproc1) <= '1'; -- signal that active processor 1 is
now faulted
        led(2) <= '1'; -- light up LED if processor 1 isn't working
        else                    -- else if none of the three active processors can agree on
the output, or at least two of them are in error mode. . .
        Nmux <= Spareproc;
        Procflag <= '1';
        Procerror <= '1';        -- in this case, none of the three active
processors agree and so there is an error
        Procfaults(Actproc1) <= '1'; -- signal that all active processors are
faulty, since at least two of them must be
        Procfaults(Actproc2) <= '1';
        Procfaults(Actproc3) <= '1';
        Procrun <= '0';          -- halt execution of the program (not
currently effective)
        led(2) <= '1';
        end if;

        if Nmux = 0 then        -- led(3) lights up when Nmux has changed,
proving that voter is doing something
        led(3) <= '0';
        else
        led(3) <= '1';
        end if;
-- to get the spare to work, you can try setting the if conditions to be true only if
Procfault(ActprocX) = '0', but then you run
-- the risk of setting these to '1' before the program starts running, therefore
always making the system run off of Spareproc
-- Currently, I believe that when two procs are faulted, it defaults to those two
since they are the same, and therefore errors.
-- So currently the spare functionality doesn't work. Also you can try
uncommenting the "Actproc1 <= Spareproc;" lines so that the Spareproc

```

```

-- is brought online when one of the original three has a fault, but I was just
having the system crash on the first fault when I did that
-- To fix it would require a bit of thought. So as of now, the system is TMR but
not TMR+spare.
end process;

-- assign outputs from voter instantaneously as soon as Nmux is changed
ahbmo(0) <= ahbmo_signal(Nmux);
irqo(0) <= irqo_signal(Nmux);
dbggo(0) <= dbggo_signal(Nmux);

-- introducing an error into the data stream by pressing pushbuttons
ahbmi_hrdata0 <= ahbmi.hrdata(AHBDW-1 downto AHBDW-4) & (button(1)
OR ahbmi.hrdata(AHBDW-5)) & ahbmi.hrdata(AHBDW-6 downto 0);
ahbmi_hrdata1 <= ahbmi.hrdata(AHBDW-1 downto AHBDW-4) & (button(2)
OR ahbmi.hrdata(AHBDW-5)) & ahbmi.hrdata(AHBDW-6 downto 0);
ahbmi_hrdata2 <= ahbmi.hrdata(AHBDW-1 downto AHBDW-4) & (button(3)
OR ahbmi.hrdata(AHBDW-5)) & ahbmi.hrdata(AHBDW-6 downto 0);
ahbmi_hrdata3 <= ahbmi.hrdata(AHBDW-1 downto AHBDW-4) & (button(4)
OR ahbmi.hrdata(AHBDW-5)) & ahbmi.hrdata(AHBDW-6 downto 0);

ahbmi_signal(0) <= (hgrant => ahbmi.hgrant, hready => ahbmi.hready, hresp =>
ahbmi.hresp, hrdata => ahbmi_hrdata0, hirq => ahbmi.hirq,
testen => ahbmi.testen, testrst => ahbmi.testrst, scanen =>
ahbmi.scanen, testoen => ahbmi.testoen, testin => ahbmi.testin);
ahbmi_signal(1) <= (hgrant => ahbmi.hgrant, hready => ahbmi.hready, hresp =>
ahbmi.hresp, hrdata => ahbmi_hrdata1, hirq => ahbmi.hirq,
testen => ahbmi.testen, testrst => ahbmi.testrst, scanen =>
ahbmi.scanen, testoen => ahbmi.testoen, testin => ahbmi.testin);
ahbmi_signal(2) <= (hgrant => ahbmi.hgrant, hready => ahbmi.hready, hresp =>
ahbmi.hresp, hrdata => ahbmi_hrdata2, hirq => ahbmi.hirq,
testen => ahbmi.testen, testrst => ahbmi.testrst, scanen =>
ahbmi.scanen, testoen => ahbmi.testoen, testin => ahbmi.testin);
ahbmi_signal(3) <= (hgrant => ahbmi.hgrant, hready => ahbmi.hready, hresp =>
ahbmi.hresp, hrdata => ahbmi_hrdata3, hirq => ahbmi.hirq,
testen => ahbmi.testen, testrst => ahbmi.testrst, scanen =>
ahbmi.scanen, testoen => ahbmi.testoen, testin => ahbmi.testin);

-----
--- LEON3 processor and DSU -----
-----

nosh : if CFG_GRFPUISH = 0 generate
cpu : for i in 0 to num_procs-1 generate

```

```

l3s : if CFG_LEON3FT_EN = 0 generate
  u0 : leon3s -- LEON3 processor
  generic map (0, fabtech, memtech, CFG_NWIN, CFG_DSU, CFG_FPU, CFG_V8,
-- 0 was i
  0, CFG_MAC, pclow, CFG_NOTAG, CFG_NWP, CFG_ICEN, CFG_IREPL,
CFG_ISETS, CFG_ILINE,
  CFG_ISETSZ, CFG_ILOCK, CFG_DCEN, CFG_DREPL, CFG_DSETS,
CFG_DLINE, CFG_DSETSZ,
  CFG_DLOCK, CFG_DSNOOP, CFG_ILRAMEN, CFG_ILRAMSZ,
CFG_ILRAMADDR, CFG_DLRAMEN,
  CFG_DLRAMSZ, CFG_DLRAMADDR, CFG_MMUEN, CFG_ITLBNUM,
CFG_DTLBNUM, CFG_TLB_TYPE, CFG_TLB_REP,
  CFG_LDDEL, disas, CFG_ITBSZ, CFG_PWD, CFG_SVT, CFG_RSTADDR,
CFG_NCPU-1, -- 0 was CFG_NCPU-1
  CFG_DFIXED, CFG_SCAN, CFG_MMU_PAGE, CFG_BP)
  port map (clk, rstn, ahbmi_signal(i), ahbmo_signal(i), ahbsi, ahbso,
  irqi(0), irqo_signal(i), dbgi(0), dbgo_signal(i));
-- port map (clk, rstn, ahbmi, ahbmo(i), ahbsi, ahbso, -- this is the original port
map
-- irqi(i), irqo(i), dbgi(i), dbgo(i));
  end generate;
end generate;
end generate;

gpio0 : if CFG_GRGPIO_ENABLE /= 0 generate -- GPIO unit
  grgpio0: grgpio
  generic map(pindex => 10, paddr => 10, imask => CFG_GRGPIO_IMASK, nbits =>
7)
  port map(rst => rstn, clk => clk, apbi => apbi, apbo => apbo(10),
  gpioi => gpioi, gpioo => gpioo);
  pio_pads : for i in 0 to 2 generate
    pio_pad : iopad generic map (tech => padtech, level => cmos, voltage => x25v)
    port map (switch(i), gpioo.dout(i), gpioo.oen(i), gpioi.din(i));
  end generate;
  -- the following section was commented out to allow the pushbuttons to be
used for error insertion
  -- pio_pads2 : for i in 3 to 5 generate
  -- pio_pad : inpad generic map (tech => padtech, level => cmos, voltage => x15v)
  -- port map (button(i-2), gpioi.din(i));
  -- end generate;
end generate;

```