Project Number: BS2-1102

MODULAR REDUNDANCY FOR ROBUST SOFT IP-CORES

A Major Qualifying Project Report
submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the
Degree of Bachelor of Science

By

Gary Katzoff

April 26, 2012

Approved:

_____
Professor Berk Sunar, Advisor

TABLE OF CONTENTS

Figures iii

Abstract iv

Acknowledgements v

Executive Summary vi

1. Framing the Problem 1

1.1 Introduction 1

1.2 Project Goals 2

2. Background Information 3

2.1 Field-Programmable Gate Array 3

2.2 Soft IP-Cores 3

2.3 Triple Modular Redundancy 3

2.4 Lockstep 4

2.5 General Dynamics C4 Systems 4

2.6 Altera Corporation 4

2.6.1 Nios II 4

2.6.2 Quartus II 5

3. Tackling the Problem 6

3.1 Testing the Board 6

3.2 Tutorial 7

3.3 Determining System Requirements 8

3.4 Hex File 9

3.5 Adding Lockstep Components 10

3.6 Removing the Debug Module 11

3.7 Changes to Create a Multiple CPU System 11

3.8 Unifying the Memory Address Span 13

3.9 Adding the Reset Source 15

3.10 Adding the Data Conditions 16

3.11 Error Injection 16
3.12 readdata Error Injection 17
3.13 Adding the Third CPU 18
3.14 Error Injection in the Three CPU System 20
3.15 Resolving the LED/Reset Issue 20
3.16 Reinstating the Internal Reset 22
3.17 Debounce Circuit 22
4. Further Enhancements 23
4.1 Debounce Circuit 23
4.2 Clock Skew and Interrupts 23
4.3 Intelligent Design 23
4.4 Linux 25
5. Bibliography 26
6. Appendices 27
Appendix A. unified_lockstep.v 27

FIGURES

Figure 3.1: Altera Embedded Systems Development Kit, Cyclone III Edition 6
Figure 3.2: Nios II Hardware Development Tutorial System 8
Figure 3.3: Dual Lockstep System with Single CPU and RAM 12
Figure 3.4: Unified Lockstep Bridge System 14

ABSTRACT

This project successfully implements a triple modular redundant system on an Altera field-programmable gate array, FPGA, development board for General Dynamics C4 Systems. The system implements a simple counting program simultaneously on three Altera Nios II soft IP-core CPUs; and has an error detecting voting scheme to catch errors, disable faulty CPUs, pass through good signals between the CPUs and the peripherals, and reset the system if it is compromised.

ACKNOWLEDGEMENTS

I would like to thank Professor Berk Sunar for his support and advising. Also, I would like to thank Brendon Chetwynd and Gerardo Orlando of General Dynamics C4 Systems for supplying a workspace, equipment, and helping to shape the project's direction. To Joyce Willette, William Moyer, and Rodney Frazer of the Altera Corporation, thank you for your patience and assistance in both providing the development kit and aiding in the various tribulations of the project.

EXECUTIVE SUMMARY

The project goal was to create a triple modular redundant system using three Altera Nios II soft IP-cores on a FPGA development board for General Dynamics C4 Systems at its Needham, Massachusetts location. The system needed to be implemented in lockstep so that a voting scheme can catch errors from the redundant CPUs, disable the CPU with the errors, and only pass through good signals from the remaining CPUs.

After receiving the FPGA development kit from Altera, its functionality was tested with a simple logic design. After the success of this simple hardware counter design, an Altera tutorial, Nios II Hardware Development Tutorial, was used to implement a basic Nios II system in the FPGA. It was then decided that the tutorial would be used as the basis of for the triple modular redundant system design.

Before working on adding the addition CPUs and the voting scheme, the debug module needed to be removed from the Nios II, since the development tools cannot setup the system to debug, or program, multiple CPUs simultaneously. Instead, the program needed to be added to the hardware design, by converting it into a hex file. Using a hex file allowed the program to begin running as soon as the system started.

With a test of using a hex file in the tutorial system, next the voting scheme and lockstep needed to be added. Two lockstep bridges were added to the system, one to handle the Nios II's data master, and one for the instruction master. The system went through several failed designs with varying numbers of CPUs and both a single on-chip RAM, and two dedicated to the data and instructions respectively.

Finally, to resolve dual RAM and memory addressing issues, the memory was permanently restored to a single on-chip RAM and the lockstep bridges were combined into a

single unified lockstep bridge. With the issues resolved, the voting scheme and error detection was implemented for two CPUs. After the error detection circuit was designed and implemented it needed to be tested, by forcing errors.

Initially, the injected errors caused simulated CPU errors. After getting the simulated error to be detected, and the 'faulty' CPU disabled, real errors were injected into the CPUs. With some difficulty the detection of real errors and disabling of CPUs worked with some reset issues.

Finally, the third and final CPU was added to the system for full triple modular redundancy, and the voting scheme and error detection where modified to accommodate the new situation. This system worked, but had some peculiar reset issues. The reset issues were eventually resolved, including the one that occurred when the internal reset was added to the error detection design.

The triple modular redundant system, with a voting scheme and error detection in lockstep, reached its goal. However, the system is not perfect, and there are step which can be taken to improve upon the system.

1. FRAMING THE PROBLEM

1.1 INTRODUCTION

This Major Qualifying Project, MQP, was conducted off campus at the sponsor General Dynamics C4 Systems' Needham, Massachusetts facilities. The MQP team was a single individual, who commuted to C4 Systems every day during the B Term 2011. At GDC4S, the workstation was a cubicle within an unclassified section of one of the facility's buildings.

Along with Professor Sunar, there were two GDC4S advisors Brendon Chetwynd and Gerardo Orlando. Brendon and Gerardo brainstormed the initial project suggestion. The general objective was to create a triple modular redundant system which used soft IP-cores in lockstep. This topic was suggested due to the MQP team's interest in pursuing a hardware project using a field-programmable gate array, FPGA, development board. Depending on how the project progressed there was flexibility in the object, as well as potential additional objectives, which included adding Linux to the system.

Motivation for this project came from multiple places. For the MQP team, motivation came from wanting to work on a digital hardware project, preferably one which used a FPGA development board and either the hardware description language VHDL or Verilog. As for General Dynamics C4 Systems, very little is known about its motivation. It was indicated that Brendon and Gerardo were interested in seeing a system such as this one implemented entirely on a FPGA. Also, it was noted that neither of them had experience with Altera's current development environment, so they were using this project as a test of the tools. It is possible that this project has some direct correlation to a current or future GDC4S project, but since both Brendon and Gerardo work in a classified department it was never indicated that this project had any connections to current or future C4 Systems products.

1.2 PROJECT GOALS

This project had a number of goals and aspirations, all of which were flexible objectives based on how likely they were to be accomplished by a one person team who had to learn about the development environment, traverse unforeseen pitfalls, and other variables in seven weeks. The main objective was to create a triple modular redundant system which used soft IP-cores in lockstep on a FPGA development board. The hope was that the final system, at the end of B Term 2011, would have the system up to this main objective, but it was understood that depending on various circumstances that the system may not reach this objective. Additional goals included the potential of running Linux in the system and/or getting to the point where the system can correct itself.

2. BACKGROUND INFORMATION

2.1 FIELD-PROGRAMMABLE GATE ARRAY

A field-programmable gate array, FPGA, is an array of logic gates which can be programmed to alter their connections to create various logic designs. A FPGA provides a powerful development platform which allows designers to 'program' hardware in hardware description languages, HDLs, such as VHDL and Verilog, and realize the designs by programming the FPGA on its development board. Programming FPGAs on development boards allows for quick modification and corrections to a logic design, making FPGAs an ideal development platform.

2.2 SOFT IP-CORES

Soft IP-cores are microprocessor architectures coded in a hardware description language for use in a FPGA. Such cores allow designers more flexibility than physical CPUs. Unlike their physical counterparts, soft IP-cores can be integrated directly into designs instead of having to send signals in and out of the FPGA to communicate with the external chip, which can potentially lead to signal delays. Also, depending on the intellectual property, IP, license one has with the soft IP-core's developer the core can be customized to varying degrees to fit the varying needs to the system, which can include access to the core's source code.

2.3 TRIPLE MODULAR REDUNDANCY

Triple modular redundancy, TMR, is a simple method for trying to improve the reliability of a system built out of low reliability components. TMR uses a simple majority rule, two to one majority, to decide if a given copy of the triplicate component or system has an error. In the case of this project, the CPU is the triple modular redundant component of the overall system.

2.4 LOCKSTEP

Being in lockstep is about synchronicity. All of the components need to be synced. Lockstep is achieved by clocking the actions of the components, and by advancing all components at the same step in the overall assignment that they are performing. In particular, any simultaneously running redundant components, such as triple modular redundant CPUs, need to be set up so that they are performing the exact same task synchronously in lockstep.

2.5 GENERAL DYNAMICS C4 SYSTEMS

General Dynamics C4 Systems, GDC4S, is a subdivision of General Dynamics which is a "leading provider of network-centric solutions" [1]. Those "leadership credentials come from applying world-class capabilities to create high-value, low risk solutions for use on land, at or under the sea, in the air and in space" [1]. Most of GDC4S's contracts are with government and defense customers.

2.6 ALTERA CORPORATION

Altera provides custom logic solutions for "customers in a wide variety of industries, including automotive, broadcast, computer and storage, consumer, industrial, medical, military, test and measurement, wireless, and wireline" [2]. It also offers "fully integrated software development tools, versatile embedded processors, optimized intellectual property (IP) cores, reference designs examples, and a variety of development kits" [2].

2.6.1 NIOS II

The Altera Nios II is a 32-bit soft IP-core embedded processor for use in Altera FPGAs and ASICs. This processor comes in versions, economy Nios II/e, standard Nios II/s, and fast Nios II/f. Both the Nios II/s and Nios II/f were used in developing the system. The final design

features the high performance Nios II/f, which has the potential to run some versions of Linux [3] [4].

2.6.2 QUARTUS II

Quartus II is an Altera development environment used to design and synthesize logic designs for Altera FPGAs and other programmable devices. While Quartus II is the main design environment, it employs a suite of included programs and tools, such as Qsys, Eclipse, and SignalTap [5]:

Qsys is a GUI-based program which allows the user to interconnect system components together and optimizes their connections. It also allows for the modification of some components such as the Nios II processor and the generation and addition of custom user created components [5].

Eclipse is a software development program, and this particular version of Eclipse has been customized by Altera to include Nios II development tools.

SignalTap is a tool used to study signals of a running logic design. It is used to capture signal values at and around a given trigger signal. The captured data can be used to prove the signal behavior properly, or discover the location and source of an error.

3. TACKLING THE PROBLEM

3.1 TESTING THE BOARD

The work truly began after receiving the Altera Embedded Systems Development Kit, Cyclone III Edition, see Figure 3.1, beginning with a simple test to see if the development board worked. The design is based on a simple VHDL design from an ECE 3810 lab. This design takes the 50MHz system clock and builds a one hertz clock from it. A counter counts from 0 to 15 and resets to 0 based upon the one hertz clock. The counter's current value is displayed on the system's LEDs in binary. Since there are eight LEDs on the board, and the counted value only uses four bits, the LEDs were setup to display the counter's value twice. In addition, at any point, if the push button designated as the reset was triggered the counter reset to 0 until the push button was released allowing the counter to continue counting up from 0.

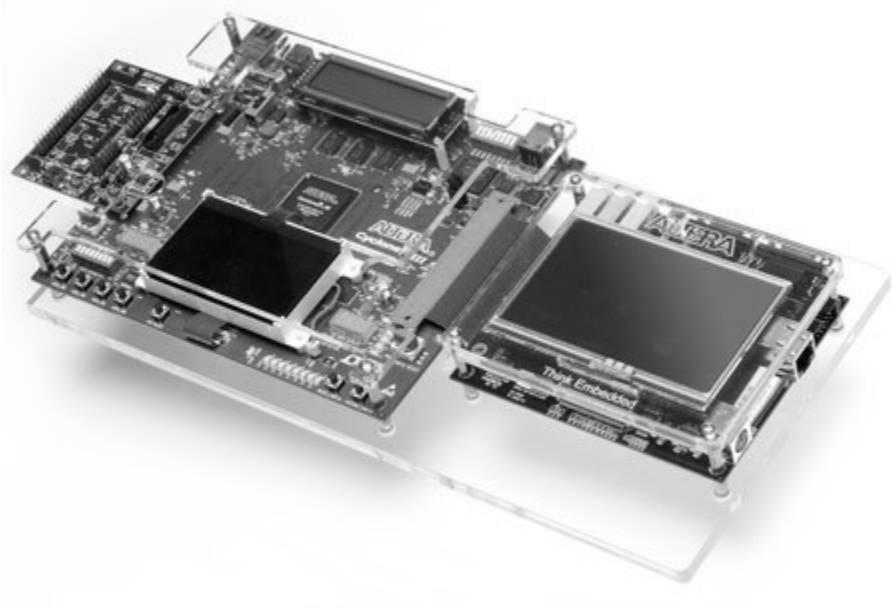

Figure 3.1: Altera Embedded Systems Development Kit, Cyclone III Edition [6]

Implementing this counter on the newly installed development board, for reasons unknown, did not appear to work, yet the LEDs were all lit up. As it turns out, it worked fine. As it turns out the push buttons on the development board were active low not active high. If

one pushed down the push button designated as the reset for the system, the LEDs went through a count sequence. To fix this issue one simply had to replace the '1' for reset in the code for '0' and the reset worked fine. Apparently the LEDs had a similar situation. The reason why they had all been lit up was because the system was in a constant state of reset, and in reset the counter restarts at zero; and therefore all of the LEDs were set to low. Since low is active and high is not the code had all of the LED's binary inverted. After changing all of the '1's to '0's and vice versa, the system worked as had initially been envisioned.

3.2 TUTORIAL

After checking that the board was functional, the next step was to create a design which used a single soft IP-core microprocessor, in particular a version of Altera's Nios II microprocessor. Altera has a tutorial called Nios II Hardware Development Tutorial, see Figure 3.2. The tutorial shows how to make a simple Nios II microprocessor system in hardware and how to add some simple software to run upon it. Using a Nios II, on-chip memory, the system clock, JTAG UART, system clock timer, system ID, and the LEDs along with supplied C code, the tutorial's system created a software based counter, similar to the hardware count used to test the board. This counter counts from 0 to 255. While counting, the eight LEDs display the binary value of the current count, and the console of Eclipse displays the results simultaneously in hexadecimal, 00 to ff. Also, at the beginning of each iteration of the count a border of asterisks surround some text announcing the programs intent is printed in the console, which is subsequently followed by the printing of the hexadecimal values. For aesthetic purposes, a tilde, '~,' was added to the beginning of the line of C code which controls the values sent to the LED. The tilde inverted the value so that the active low LEDs lit up for '1's instead of '0's.

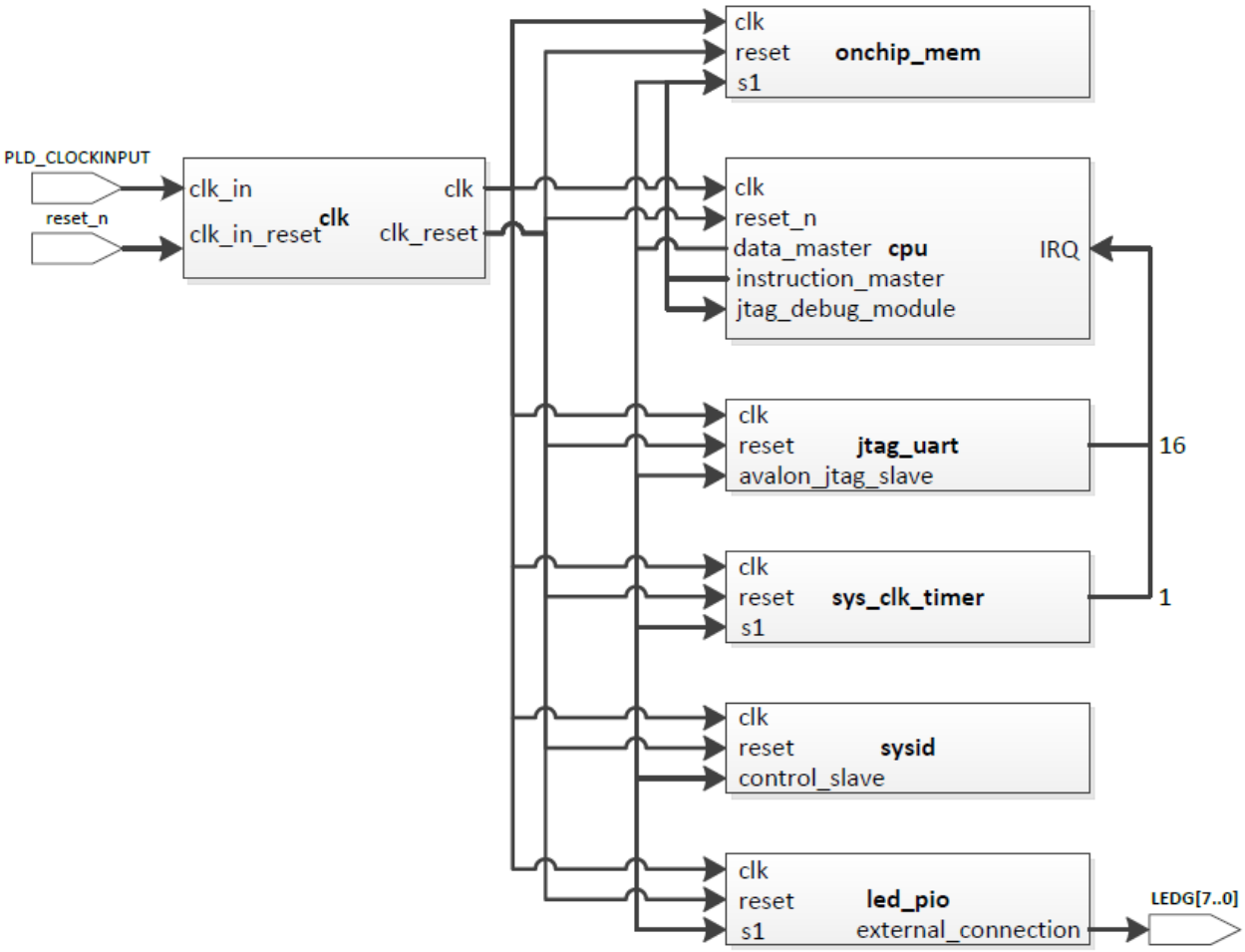

Figure 3.2: Nios II Hardware Development Tutorial System

3.3 DETERMINING SYSTEM REQUIREMENTS

With the tutorial's single CPU system functional, the next step was creating a redundant system. In the multiple CPU system, all of the CPUs talk to the same peripherals; but since each peripheral can only handle being the slave to one master, an intermediary component is needed to mediate communications between the slaves, peripherals, and the multiple masters, CPUs. Such an intermediary component does not exist in the standard Altera libraries, so a custom component would be necessary to accomplish the task.

It was determined that the best course of action would be to start with setting up the full triple CPU system, but running it as though it was only the single CPU tutorial. The full system

would be setup with the three identical CPUs, all of the necessary peripherals, and whatever custom components which were needed to implement the lockstep. Initially, the custom components would be setup as a pass-through for the instruction and data masters of the first CPU, with the other two CPUs disabled, placing them into a constant wait state using the `waitrequest` signals. Once it was confirmed that the signals of the first CPU went through, the second CPU would be enabled and a form of the lockstep voting scheme would be implemented. Subsequently, with two CPUs working the third would be added and the voting scheme would be updated. In both of these multiple CPU systems, one, or more, push buttons would be tasked with injecting errors into the data or instruction masters of the CPUs to simulate actual errors and the voting schemes reaction.

3.4 HEX FILE

Qsys does not allow for the simultaneous debugging of multiple microprocessors, therefore none of the three CPUs were setup with a debug module. With the removal of the debug module from the microprocessors' design, it was decided that it would be best if Eclipse was not needed to install the software on the hardware design. To accomplish such a task the programming was saved as a hex file. Hex files are files that use hexadecimal values to initialize memory components. If the hex file has executable code initialized, the system will automatically start running it at start up instead of having to wait for the software to be downloaded manually via Eclipse.

After creating the hex file for the tutorial system, and adding it to the hardware design, the FPGA was programmed with the new system design with this preinstalled software. At first the system failed to work, which was due to an aspect of the tutorial. Now that the software was not being downloaded from Eclipse, the program was not using Eclipse's console to display the

welcome message and count. Not being connected to the console was a problem because the BSP, Board Support Package, was setup to utilize reduced device drivers; meaning that the program would remain paused until it recognized that it was connected to all of its necessary components, which included a console. There were two options to easily work around this problem, either change the BSP setting so that the program could run without the use of a console or to open up a console elsewhere. The latter option was chosen. Using the Nios II 11.0sp1 Command Shell one can open up a console, or terminal, and the program ran as it had before, but this time it worked independent of Eclipse.

3.5 ADDING LOCKSTEP COMPONENTS

With the hex file working on the single CPU tutorial's system, the next step was to actually add the additional CPUs and the custom components. Using parts from a sample design thrown together by the Altera Embedded Specialist assisting with the project, an initial design was setup. This design used two custom components, data_lockstep and instruction_lockstep, and two 20 kilobyte on-chip RAM, name data_ram and instruction_ram. Each lockstep custom component had three slaves for three CPUs' data or instruction masters, respectively, and one master to connect to the peripherals. The data_ram and instruction_ram are slaves to their namesake lockstep components.

When compiling the software in Eclipse, an error occurred informing that instruction_ram needed another 31,916 bytes. Modifying the instruction_ram to 60 kilobytes removed the error and allowed the software to fully compile. This request for a larger capacity for the instruction_ram was rather odd since the tutorial's single on-chip RAM was a total of 20 kilobytes; and the selection of 20 kilobytes for both the data_ram and instruction_ram was to make sure each was of a sufficient capacity.

However, while the software compiled and the hex file was added to the design, the system would not run, even with an open terminal. To resolve this issue, the system was rebuilt from scratch. The system was rebuilt to match the original tutorial system, but with some minor modifications. This system used the Nios II/f, with a debug module, from the custom component sample, instead of the Nios II/s of the tutorial. The system was first run by downloading the software through Eclipse, and then by running it from a hex file. Both of these methods worked correctly.

3.6 REMOVING THE DEBUG MODULE

The next step was to remove the debug module and run the system again, before adding the additional CPUs and the lockstep bridges. Running this system caused a break vector memory error, because the CPU's break vector was still pointing to the nonexistent debug module. Setting the break vector memory option to onchip_mem.s1 resolved this issue and allowed the system to run without a debug module.

3.7 CHANGES TO CREATE A MULTIPLE CPU SYSTEM

With a working debug module free single CPU system, the next step was to add the lockstep bridges and additional CPUs. Problems occurred at this point. Several system variations were tested. These designs included the data and instruction lockstep bridges with respective RAM, with one on-chip RAM, see Figure 3.3, with three CPUs with both one and two RAM, and even a single CPU setup with the dual RAM design. None of these systems worked, which lead to the belief that a problem lay with both the lockstep bridges and the two RAM setup.

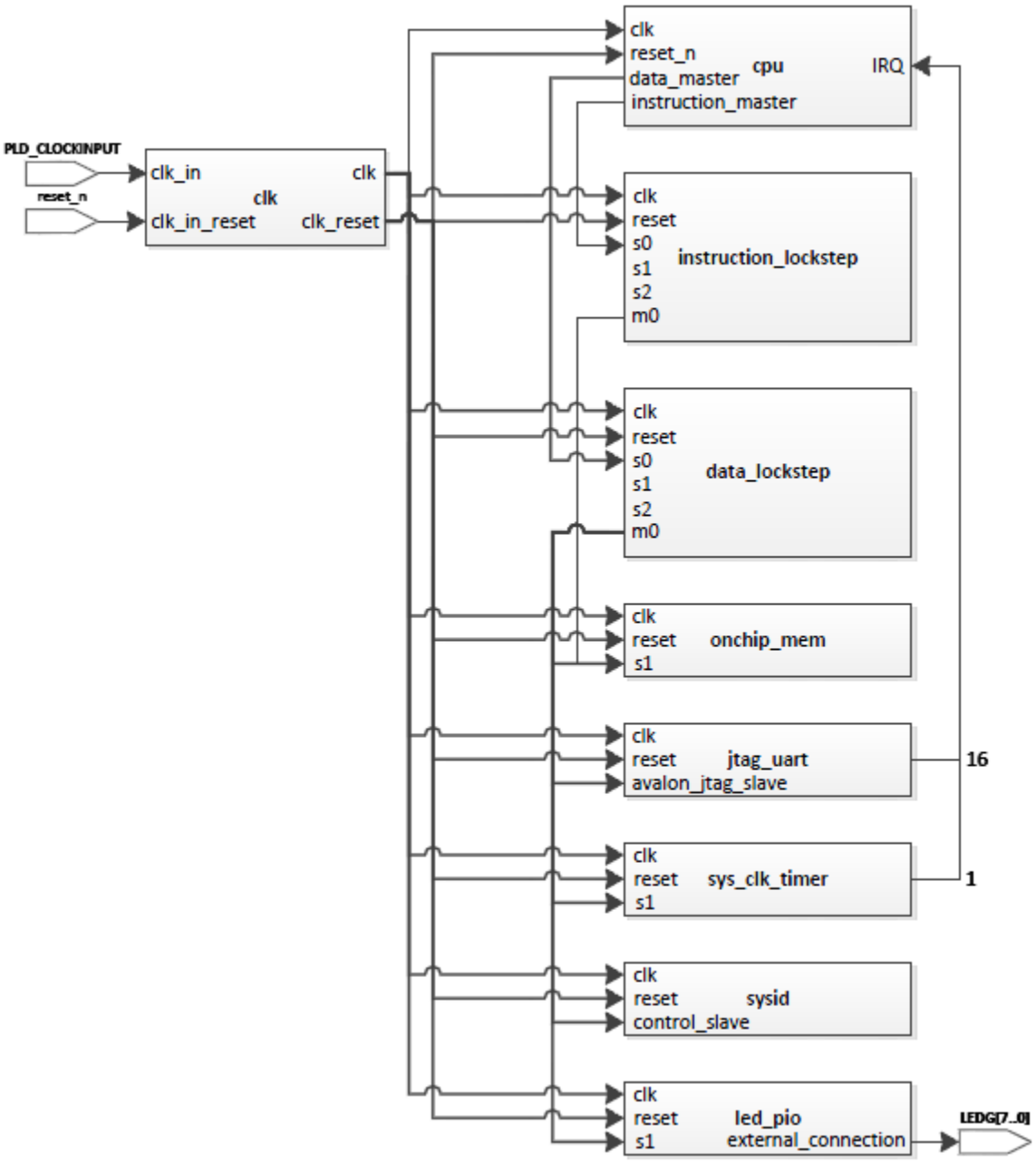

Figure 3.3: Dual Lockstep System with Single CPU and RAM

As a temporary step, the lockstep bridges were replaced with the standard pipeline bridges in the single CPU, single RAM design, which simply passed through the CPU's data and instruction master to the peripherals. This design worked, so the two RAM design was implemented, and a correction for the lockstep bridges was found shortly thereafter. The line of

Verilog code which passed the slave address through as the master address needed to concatenate with the slave address with 2'b00.

This correction was followed by the addition of a second CPU was added to the system. This second CPU wanted to look at the instruction lockstep for its reset, exception, and break vectors instead of the on-chip RAM, which it refused to recognize. As it turns out the reason for this was a parameter called "bridgesToMaster" was missing from two of the slave interfaces in the component's TCL file.

3.8 UNIFYING THE MEMORY ADDRESS SPAN

During this time there was discussion and difficulty with regards to the address spans of the data and instructions in the RAM. The addresses of different slaves cannot be mapped to the same address bits. To resolve the addressing issues, it was suggested to connect the data and instruction masters of the Nios II to different slaves on the same component, which would allow the system to allow them to share the same base address, thus unifying the address ranges.

As a result of this addressing solution, a few things happened to the RAM design and the lockstep bridges. Up until this point, throughout the various designs, the on-chip RAM alternated between one on-chip RAM and two, one for data and the other for instructions. Moving forward, the on-chip RAM was only one component, see Figure 3.4.

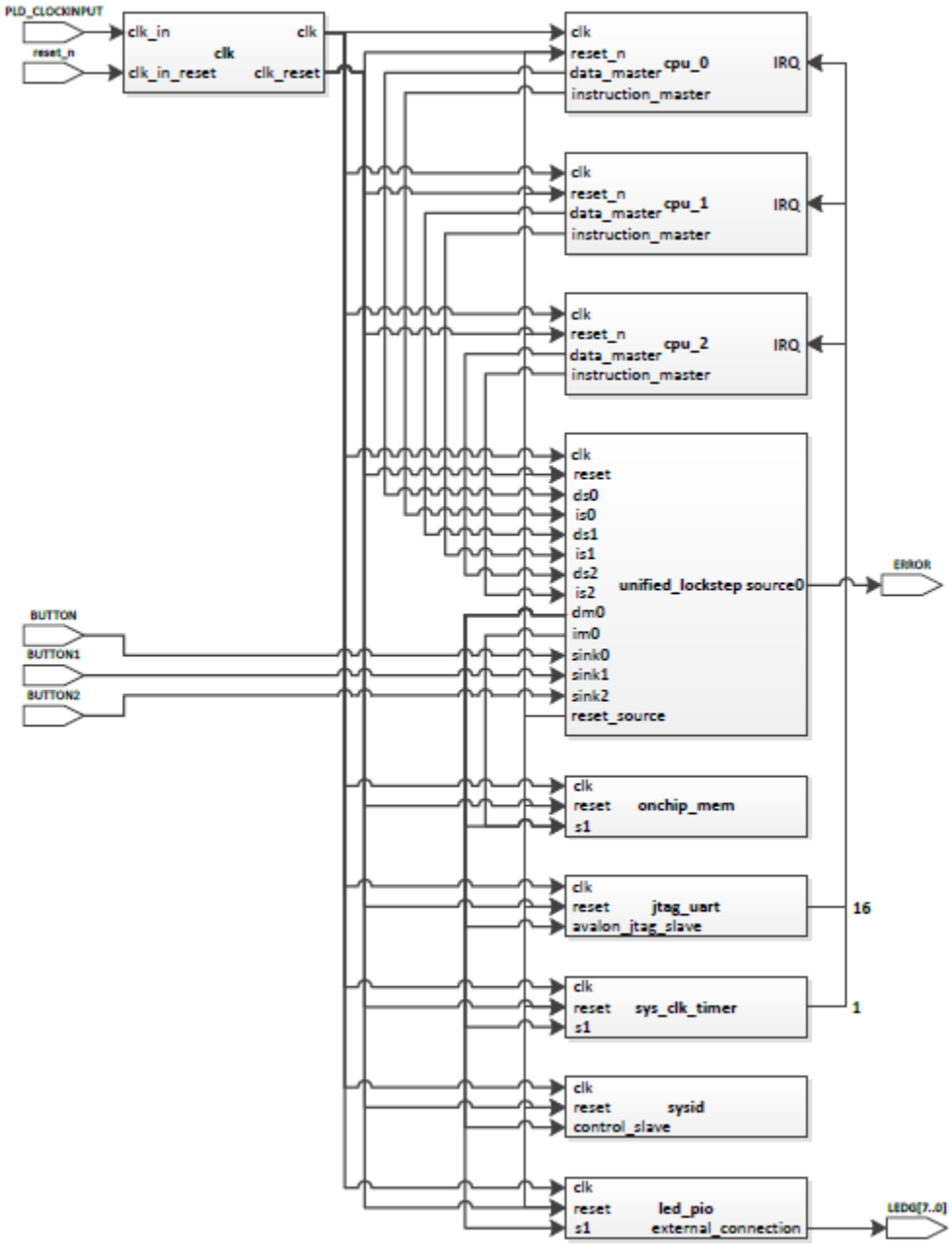

Figure 3.4: Unified Lockstep Bridge System

The lockstep bridges had a more drastic change. To accommodate the requirements of the unified addressing both the data and instruction masters of a given CPU needed to be slaves to the same component. This requirement meant that to facilitate the comparison of data and instruction masters from the various CPUs, they all needed to be in the same lockstep bridge custom component. Therefore, the data and instruction lockstep bridges were replaced with a

single unified lockstep bridge. The unified lockstep bridge was setup to support up to six slaves, three data and three instruction masters from Nios II CPUs, and two masters, data and instruction. Also, the master and slave addresses were made symmetric, which removed the need for the concatenated shifting.

3.9 ADDING THE RESET SOURCE

Implementation of the unified lockstep design, with two disabled CPUs, and just passing through the working CPU's masters worked. This test allowed for the first steps toward a voting scheme in lockstep to be created. Following the test, the second CPU was enabled and a clocked comparison of the instruction masters was setup.

At this time, an internal reset was added to the system. The idea was to allow the voting scheme to autonomously reset the system when the redundancy becomes completely compromised. A `reset_source` was added to the unified lockstep component, and was connected to all of the resets of the system's components. Now, the push button and the voting scheme would be able to reset the system.

The code controlling the reset signal from the voting scheme had some issues. When the system started up it initially went into a sort of perpetual reset state. It turned out that there was a problem with the instruction master address comparison. The addresses were comparing when the system started up, but the comparator needed to be comparing the addresses only when they are being read. Even with this fix, there were issues with the reset. Since fixing the reset was taking a while, it was decided that the internal reset would be postponed, and fully implementing the voting scheme would be given priority.

3.10 ADDING THE DATA CONDITIONS

Returning to the voting scheme, the first step was to add data master comparison conditions. Both the instruction and data conditions were compared in the same condition statement. There were concerns about whether putting all of these comparisons in the same statement would cause problems. It was determined that the statement was not big enough to be a concern, and that pipelining of the equation would not be necessary.

3.11 ERROR INJECTION

After adding the data conditions to the comparison equation, the next step was to make sure that the code worked. To test this error detection code, some modifications to the unified lockstep bridge's needed to be added. First, some method of alerting the user that an error occurred needed to be added. Since the autonomous reset was still disabled, resetting the system at the error was not an option. What was done instead was adding a source, which was connected to an LED. The idea was that when an error was detected that the LED would light up to alert the user. Since all of the eight usable LEDs where initially setup to display the value that the counting software was up to, one of these LEDs, specifically the least significant bit LED, needed to be commandeered to implement this design.

It was decided that to test the error detection circuit an error would be injected into the system. Specifically, the error was going to be added to the `writedata` signal of data slave 1, or the data master from the second CPU. The `writedata` signal was chosen because it would result in a simulated error. What is meant by simulated, is that the signal is modified by the error injection after it leaves the CPU and the error is detected by the code and removed before the peripherals. Thus, the CPU would never know of the error.

To 'inject' an error, the system needed to allow the user to manually manipulate the signal. This manipulation was achieved by adding a sink to the unified lockstep. The sink was connected to the signal coming from one of the currently unused push buttons. When the push button was pressed, the corresponding code caused an error in the `writedata` signal. Initially, the error was adding one to the signal, but it was shortly changed to the more dramatic error of inverting the signal. It must be noted that since the `writedata` signal is an input port of the unified lockstep bridge, it itself cannot be manipulated, but instead a new signal internal to the lockstep bridge was created, manipulated, and used in the comparison.

Initially, the `writedata` error detection did not seem to work, as the LED never lit up when the error was injected. The always statement which controlled the error detection went through a few iterations, which did not work successfully, until it was decided that it would be better to use a ternary operation. With the help of the Altera Field Applications Engineer, the `writedata` error injection and detection became functional. In this current system, when an error was detected the LED lit up and stays on, until the system is reset.

3.12 READDATA ERROR INJECTION

Following the success of the `writedata` error injection, the next step was to inject an error in the `readdata` signal instead. A `readdata` error is a much more substantial error. The signal `readdata` is an output signal unlike `writedata` which is an input signal in the unified lockstep. What this difference means is that an error in the `readdata` signal is sent directly to the CPU, and subsequently corrupts any number of the signals it send back to the lockstep bridge. Unlike the `writedata` error injection, the `readdata` error injection is overall a better test for the voting scheme.

The `readdata` error injection worked, with a slight problem. After detecting the error, the LED lit up just as it did in the `writedata` design, yet the LED in this case would only turn off if the FPGA board was reprogrammed or turned off. This problem was resolved by making sure the branchPredicitonType parameter for the CPUs in the Qsys file were set to "Static" instead of "Automatic". To make this correction required opening the .qsys file in a text editor outside of Qsys, finding the branchPredicitonType parameters, and correcting the parameters for the second and third CPU. The correction partially corrected the problem. Now, instead of needing to reprogram the board to clear the LED, it could be done by resetting the system. Unfortunately, the clearing of the LED required resetting the system twice. While this issue needed a resolution, it gave the system the overall functionality needed to start working on adding the third CPU to the voting scheme design.

3.13 ADDING THE THIRD CPU

After enabling the third CPU, by reengaging `waitrequest`, the addition of the third CPU was implemented in the voting scheme. Adding the CPU required new code, but luckily most of the existing voting scheme remained as a basis. However, before the code could be written two steps needed to be taken. First, the error injection was disabled allowing the focus to be solely on obtaining a triple modular redundant CPU system up and running. Second, the plan for the behavior of the system needed to be worked out. Work on the idea for how this triple CPU system would work had been developing in the background while working on the previous system iterations.

When looking at comparing two CPUs the code was rather straightforward; simply check to see if there was a difference between the various corresponding signals and send an alert to the user, such as a lit LED and disabled CPUs. Much of this idea can be used in the three CPU

system. While it is possible to compare three of each signal, one from each CPU, together and find out if they vary, a problem arises if one wishes to remove the faulty CPU from the system. By comparing all the signals together, it is not possible to tell which CPU has the problem. To resolve this issue three iterations to the signal `error_detected`, which was used in the dual CPU system, were implemented. Each copy of `error_detected` compared two CPUs, CPUs 0 and 1, CPUs 1 and 2, and CPUs 0 and 2. If CPU 0 had an error, both `error_detected01`and `error_detected02` would find a difference. Also, if any one of the `error_detected` signals was triggered, the LED which indicated an error would light up and remain lit.

An always statement would then look at these signals, and if two of them had been triggered, it would then activate the corresponding signal called `disabled#`. This disable signal would remain active until the system resets, and in that time it would disable its corresponding CPU and remove all of the CPU's signals from being ANDed or ORed with the correctly working signals from the other CPUs being passed through the voting scheme.

Disabling a CPU turned the system into a two CPU system, which the user will be able to visually identify by the lit LED. It is important the code would be able to have the dual CPU system run with any of the two CPUs, since any one of the three could be the first to be disabled. Also, additional precautions needed to be taken for when an error occurred in this system. In the disable always statement, the condition was added to make sure that both CPUs were disabled, therefore making it impossible for a single CPU system to run.

Following the plan, the scheme was setup and worked correctly. Some of the signals, such as the disable signals needed to be registers. Additionally, the conditions on the disable statement were replaced with a clock edge trigger, and the fact that ternary condition statements

could be nested was learned. The nesting of ternary conditions was very helpful in organizing and implementing the signal control always statement, which disables the CPUs and removes disabled signals from being passed through.

3.14 ERROR INJECTION IN THE THREE CPU SYSTEM

With a working triple CPU system, error injection could be reinstated. However, the error injection needed to be modified. Two additional sinks were added to the unified lockstep bridge, and connected to the two remaining push buttons. Each of the three push buttons, now associated with error injection, caused an error in the corresponding CPU's `readdata` signals.

Along with setting up the error injection, a new reset issue occurred. When the system reset after the second error detection, the restored system began with the error LED lit, and a CPU disabled. The CPU which was disabled was the CPU which had the first error injected. This problem was not exclusive to a given CPU. It did not matter which CPU had an error injected first and second, the system would always reset with an already disabled CPU, which had had the initial error.

3.15 RESOLVING THE LED/RESET ISSUE

The code was determined to be sound, which meant that the source of the reset problem was in an unanticipated behavior of the Nios II CPUs. After deciding it seemed to be a problem with the Nios II CPUs' behavior, a series of SignalTap tests were run to try and determine what exactly was the source of the problem. As it turns out, the SignalTap tests ended up showing that the CPUs were starting up identically when they are coming out of reset. Thus, the error occurs after the CPUs begin executing the software. With this knowledge the trigger was set to the `error_detected` signal which would activate shortly after reset. The change of the trigger uncovered a `writedata` mismatch. A second test at this trigger was also taken where the

trigger was at the end of the data capture, and the results along with several of the software files were passed on to the Altera Embedded Specialist.

While waiting for feedback, the internal reset was added back into the system, and added a new twist to the problem. When the system reset after the second error detection, the restored system began with the error LED lit, and a CPU disabled, just like before. But this time, the CPU which was disabled was the CPU which had the second error injected. Besides this 'transfer,' the problem persisted as before. Because this problem further complicated matters, the reset code was again disabled.

The Altera Embedded Specialist found the potential source of the problem. The problem was the Nios II's registers. When the Nios II startup code runs it does not initialize the registers to a known value. Usually, this lack of initialization is not relevant, but since these particular Nios II CPUs are synchronized having stale data in the registers contaminating the write data bus is a problem which the design cannot afford.

To solve this problem a modified version of the crt0.S, a file from the BSP directory, was used in place of the version that Eclipse generates when creating the BSP. The modification of crt0.S has the start up code set all of the registers to zero after the cache initializations and before the stack is initialized. This simple solution worked, but it requires the person regenerating the software to be very careful not to rebuild the BSP after adding the modified crt0.S.

Rebuilding the BSP always rebuilds the crt0.S, which effectively removes the modified crt0.S code. After adding the modified crt0.S to the proper directory, it was very important to then clean, not rebuild, the BSP project, clean the application project, and then build the application project. Once the project was built, the crt0.S and objdump needed to be checked to

make sure that the modified code was in both. If the code was there, then the hex file could be generated.

3.16 REINSTATING THE INTERNAL RESET

This correction worked, which meant that the autonomous internal reset could be added back in again. Unfortunately, adding the reset code restored the error, even when the software was regenerated again. It was suggested that this latest problem could be due to the human factor in the reset. Modifying the reset trigger condition to include the sink signals, so that the reset could not begin while one of the error injection push buttons was being pressed was the solution.

3.17 DEBOUNCE CIRCUIT

While this system works perfectly well, it is still possible to obtain the error. However, the error is not very common, and when it occurs it can be cleared in one or two resets, three maximum. This occasional problem was believed to be due to jitter in the signal from the push button, and a debounce circuit should resolve the issue. Unfortunately, the debounce circuit never became functional, and was disabled in the final system design.

4. FURTHER ENHANCEMENTS

4.1 DEBOUNCE CIRCUIT

While the current triple modular redundant design is functional, there is certainly room for improvements and additions from a subsequent Major Qualifying Project, MQP, team, if General Dynamics C4 Systems wishes to continue this project. First of which, would be to correct the remaining occasional reset issue. Whether this correction would be in the form of a debounce circuit, similar to the nonfunctional code currently in the unified lockstep bridge, or another previously unconsidered option is left to be seen.

4.2 CLOCK SKEW AND INTERRUPTS

Another concern that was brought up while working on the system was that of clock skew and the possible need to lockstep the interrupts. Nothing was done about clock skew or the interrupts because neither became an issue. It is possible however that depending on where this project may lead, a future MQP will need to deal with either clock skew or the need to lockstep interrupts, or possibly both.

4.3 INTELLIGENT DESIGN

While concerns such as the debounce circuit, clock skew, and trying to lockstep the interrupts are important to keep in mind, and will possibly need to be addressed, the next step in this project should most likely be adding some form of intelligence to the system. As it is now, the system is rather naïve and trusts its voting scheme too much. The problem is that the system is unaware that there is a difference between an error and a difference.

The system is setup to detect errors, but in actuality it can only detect differences. This distinction is why the system must be reset every time the dual CPU system shows an 'error.'

When comparing signals from two CPUs, it is impossible for the voting scheme to determine which, if either, is functioning correctly.

The triple modular redundant system has a better chance of finding which CPU has an error, but it is simply trusting that a basic two to one majority rule indicates that the single different CPU is in 'error.' It is theoretically possible that two CPUs simultaneously fault in the exact same way, resulting in the majority being in error, and the minority which will be disabled being the only CPU that is functioning properly. Also, two CPUs could simultaneously fault in two different ways causing differences between all three CPUs. In this situation the system will reset.

Therefore the next step should be to make the system intelligent enough to determine which CPU actually has an error. How to accomplish this task would be up to those working on the next MQP, but it was mentioned that it might be possible by having an addition Nios II CPU governing the voting scheme, which knew what the signals were supposed to be. It is possible that such a governing CPU could in actuality be another system of CPUs, even a system of CPUs comparing the comparisons of multiple triple modular redundant systems, which are all in lockstep.

A further development of improving the systems intelligence is the idea of adding the ability for the system to fix the CPU(s) in error and restore the triple modular redundancy without resetting the system. This self correction would require a further understanding of the functioning of the Nios II to correct it while the system is running. It seems that such a design would require the system to be paused to make the correction.

4.4 LINUX

Another possible next step that was discussed since the beginning of the project, which may or may not come before or after improving the systems intelligence, is an interest in have the three CPUs run a version of Linux in lockstep. To run Linux, each CPU will need a memory management unit, MMU, which can easily be added in Qsys. Along with the addition of MMUs, the systems voting scheme will most likely need to be modified to accommodate the complexity of running an operating system as complex as Linux and keeping it in lockstep.

5. BIBLIOGRAPHY

[1] General Dynamics C4 Systems, "About Us," 2012. [Online]. Available: http://www.gdc4s.com/content/detail.cfm?item=f3f25a7b-3707-4a75-b248-e1f4cbbe0b96. [Accessed 21 April 2012].

[2] Altera Corporation, "About," 2012. [Online]. Available: http://www.altera.com/corporate/crp-index.html. [Accessed 21 April 2012].

[3] Altera Corporation, "Nios II Processor: The World's Most Versatile Embedded Processor," 2012. [Online]. Available: http://www.altera.com/devices/processor/nios2/ni2-index.html. [Accessed 21 April 2012].

[4] Altera Corporation, "Nios II Processor Reference Handbook," May 2011. [Online]. Available: http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf. [Accessed 21 April 2012].

[5] Altera Corporation, "Quartus II Subscription Edition Software," 2012. [Online]. Available: http://www.altera.com/products/software/quartus-ii/subscription-edition/qts-se-index.html. [Accessed 21 April 2012].

[6] Altera Corporation, "Altera Embedded Systems Development Kit, Cyclone III Edition," 2012. [Online]. Available: http://www.altera.com/products/devkits/altera/kit-emb-dev-cyc3.html. [Accessed 21 April 2012].

6. APPENDICES

APPENDIX A. UNIFIED_LOCKSTEP.V

```
// Generated by Rodney Frazer
// Edited and Modified by Gary Katzoff

module unified_lockstep
#(
    parameter ADDR_WIDTH = 8
)(
    input  wire                         clk,                  // clock.clk
    input  wire                         reset,                // reset.reset

    input  wire  [( ADDR_WIDTH - 1 ):0] avs_ds0_address,      //   ds0.address
    input  wire                         avs_ds0_read,         //      .read
    output wire                 [31:0]  avs_ds0_readdata,     //      .readdata
    input  wire                         avs_ds0_write,        //      .write
    input  wire                 [31:0]  avs_ds0_writedata,    //      .writedata
    output wire                         avs_ds0_readdatavalid, //     .readdatavalid
    output reg                          avs_ds0_waitrequest,  //      .waitrequest
    input  wire                  [3:0]  avs_ds0_byteenable,   //      .byteenable

    input  wire  [( ADDR_WIDTH - 1 ):0] avs_is0_address,      //   is0.address
    input  wire                         avs_is0_read,         //      .read
    output wire                 [31:0]  avs_is0_readdata,     //      .readdata
    output wire                         avs_is0_readdatavalid, //     .readdatavalid
    output reg                          avs_is0_waitrequest,  //      .waitrequest

    input  wire  [( ADDR_WIDTH - 1 ):0] avs_ds1_address,      //   ds1.address
    input  wire                         avs_ds1_read,         //      .read
    output wire                 [31:0]  avs_ds1_readdata,     //      .readdata
    input  wire                         avs_ds1_write,        //      .write
    input  wire                 [31:0]  avs_ds1_writedata,    //      .writedata
    output wire                         avs_ds1_readdatavalid, //     .readdatavalid
    output reg                          avs_ds1_waitrequest,  //      .waitrequest
    input  wire                  [3:0]  avs_ds1_byteenable,   //      .byteenable

    input  wire  [( ADDR_WIDTH - 1 ):0] avs_is1_address,      //   is1.address
    input  wire                         avs_is1_read,         //      .read
    output wire                 [31:0]  avs_is1_readdata,     //      .readdata
    output wire                         avs_is1_readdatavalid, //     .readdatavalid
    output reg                          avs_is1_waitrequest,  //      .waitrequest

    input  wire  [( ADDR_WIDTH - 1 ):0] avs_ds2_address,      //   ds2.address
    input  wire                         avs_ds2_read,         //      .read
    output wire                 [31:0]  avs_ds2_readdata,     //      .readdata
    input  wire                         avs_ds2_write,        //      .write
    input  wire                 [31:0]  avs_ds2_writedata,    //      .writedata
    output wire                         avs_ds2_readdatavalid, //     .readdatavalid
    output reg                          avs_ds2_waitrequest,  //      .waitrequest
    input  wire                  [3:0]  avs_ds2_byteenable,   //      .byteenable

    input  wire  [( ADDR_WIDTH - 1 ):0] avs_is2_address,      //   is2.address
    input  wire                         avs_is2_read,         //      .read
    output wire                 [31:0]  avs_is2_readdata,     //      .readdata
    output wire                         avs_is2_readdatavalid, //     .readdatavalid
    output reg                          avs_is2_waitrequest,  //      .waitrequest

    output reg   [( ADDR_WIDTH - 1 ):0] avm_dm0_address,      //   dm0.address
    output reg                          avm_dm0_read,         //      .read
    input  wire                         avm_dm0_waitrequest,  //      .waitrequest
    input  wire                 [31:0]  avm_dm0_readdata,     //      .readdata
    output reg                          avm_dm0_write,        //      .write
    output reg                  [31:0]  avm_dm0_writedata,    //      .writedata
    input  wire                         avm_dm0_readdatavalid, //     .readdatavalid
    output reg                   [3:0]  avm_dm0_byteenable,   //      .byteenable
```

```
    output reg   [( ADDR_WIDTH - 1 ):0] avm_im0_address,       //   im0.address
    output reg                          avm_im0_read,          //      .read
    input  wire                         avm_im0_waitrequest,   //      .waitrequest
    input  wire                 [31:0]  avm_im0_readdata,      //      .readdata
    input  wire                         avm_im0_readdatavalid, //      .readdatavalid
    //Ports added by Gary Katzoff
    input  wire                         asi_sink0_data,        // sink0.data
    input  wire                         asi_sink1_data,        // sink1.data
    input  wire                         asi_sink2_data,        // sink2.data
    output wire                         aso_source0_data,      // source0.data

    output reg                          reset_source           // reset_source.reset
//Please note that originally all ports were wires.
);
// Some iterations of old code are commented out to show the original code of the file,
// or other options for what one can do.  It should all be labelled accordingly.

//
// data bridge
//

// these signals are simply passed thru from the master interface to the slave interfaces
//assign avs_ds0_waitrequest = avm_dm0_waitrequest;//Original
//assign avs_ds0_readdata = avm_dm0_readdata;//Original
assign avs_ds0_readdata = asi_sink0_data ? avm_dm0_readdata : ~avm_dm0_readdata;//Readdata error
injection
//assign avs_ds0_readdata = button0 ? avm_dm0_readdata : ~avm_dm0_readdata;//Debounce circuit
version
assign avs_ds0_readdatavalid = avm_dm0_readdatavalid;
//assign avs_ds1_waitrequest = avm_dm0_waitrequest;//Original
//assign avs_ds1_waitrequest = 1'b1;//Used to disable cpu_1 while testing the one CPU system
//assign avs_ds1_readdata = avm_dm0_readdata;//Original
assign avs_ds1_readdata = asi_sink1_data ? avm_dm0_readdata : ~avm_dm0_readdata;//Readdata error
injection
//assign avs_ds1_readdata = button1 ? avm_dm0_readdata : ~avm_dm0_readdata;//Debounce circuit
version
assign avs_ds1_readdatavalid = avm_dm0_readdatavalid;
//assign avs_ds2_waitrequest = avm_dm0_waitrequest;//Original
//assign avs_ds2_waitrequest = 1'b1;//Used to disable cpu_2 while testing the one and two CPU
systems
//assign avs_ds2_readdata = avm_dm0_readdata;//Original
assign avs_ds2_readdata = asi_sink2_data ? avm_dm0_readdata : ~avm_dm0_readdata;//Readdata error
injection
//assign avs_ds2_readdata = button2 ? avm_dm0_readdata : ~avm_dm0_readdata;//Debounce circuit
version
assign avs_ds2_readdatavalid = avm_dm0_readdatavalid;

//// these signals should be compared for proper lock step operation
////assign avm_dm0_writedata = avs_ds0_writedata | avs_ds1_writedata |
avs_ds2_writedata;//Original
//assign avm_dm0_writedata = avs_ds0_writedata;//Used testing the one and two CPU systems
////assign avm_dm0_address = avs_ds0_address | avs_ds1_address | avs_ds2_address;//Original
//assign avm_dm0_address = avs_ds0_address;//Used testing the one and two CPU systems
////assign avm_dm0_write = avs_ds0_write & avs_ds1_write & avs_ds2_write;//Original
//assign avm_dm0_write = avs_ds0_write;//Used testing the one and two CPU systems
////assign avm_dm0_read = avs_ds0_read & avs_ds1_read & avs_ds2_read;//Original
//assign avm_dm0_read = avs_ds0_read;//Used testing the one and two CPU systems
////assign avm_dm0_byteenable = avs_ds0_byteenable | avs_ds1_byteenable |
avs_ds2_byteenable;//Original
//assign avm_dm0_byteenable = avs_ds0_byteenable;//Used testing the one and two CPU systems


//
// instruction bridge
//

// these signals are simply passed thru from the master interface to the slave interfaces
//assign avs_is0_waitrequest = avm_im0_waitrequest;//Original
assign avs_is0_readdata = avm_im0_readdata;
assign avs_is0_readdatavalid = avm_im0_readdatavalid;
//assign avs_is1_waitrequest = avm_im0_waitrequest;//Original
```

```
//assign avs_is1_waitrequest = 1'b1;//Used to disable cpu_1 while testing the one CPU system
assign avs_is1_readdata = avm_im0_readdata;
assign avs_is1_readdatavalid = avm_im0_readdatavalid;
//assign avs_is2_waitrequest = avm_im0_waitrequest;//Original
//assign avs_is2_waitrequest = 1'b1;//Used to disable cpu_2 while testing the one and two CPU
systems
assign avs_is2_readdata = avm_im0_readdata;
assign avs_is2_readdatavalid = avm_im0_readdatavalid;

//// these signals should be compared for proper lock step operation
////assign avm_im0_address = avs_is0_address | avs_is1_address | avs_is2_address;//Original
//assign avm_im0_address = avs_is0_address;//Used while testing the one and two CPU systems
////assign avm_im0_read = avs_is0_read & avs_is1_read & avs_is2_read;//Original
//assign avm_im0_read = avs_is0_read;//Used while testing the one and two CPU systems

//// Writedata Error Injector
//wire [31:0] ds1_writedata;//A variable was needed in the writedata error injection design,
//because the writedata signals are inputs and cannot be modified directly.
////always @ (posedge clk)//initial design
////begin
////  if (asi_sink0_data == 1'b0)
////     ds1_writedata = ~avs_ds1_writedata;
////  else
////     ds1_writedata = avs_ds1_writedata;
////end
//
//assign ds1_writedata = asi_sink0_data ? avs_ds1_writedata : ~avs_ds1_writedata;//Version 2,
does not have a clocking issue

//Error/Difference Detector
//wire error_detected;//Detects an error/difference between cpu_0 and cpu_1 slave input signals
////Original fullfledged detector; error_detected01,02, and 12 are based off this design.
//assign error_detected = ((avs_is0_read != avs_is1_read) ||
//                        (avs_is0_read && (avs_is0_address != avs_is1_address)) ||
//                        (avs_ds0_read != avs_ds1_read) || (avs_ds0_write != avs_ds1_write) ||
//                        ((avs_ds0_read || avs_ds0_write) && (avs_ds0_address != avs_ds1_address))
//                        || ((avs_ds0_read || avs_ds0_write) && (avs_ds0_byteenable !=
avs_ds1_byteenable))
//                        || (avs_ds0_write && (avs_ds0_writedata != avs_ds1_writedata)));

//always @ (posedge clk)//Original Two CPU Design
//begin
//  if (reset)
//     begin
//        aso_source0_data = 1'b1;
//        reset_source <= 1'b0;
//     end
//  else
//     begin//Several conditions were tried, and worked, but the complexity increased and then
error_detected was created.
////         if (~(asi_sink0_data & ~(avs_is0_address == avs_is1_address)))//Original
////         if ((asi_sink0_data == 1'b0) || (avs_is0_address != avs_is1_address))//Same, yet
different
////         if (( asi_sink0_data == 1'b0 ) || (avs_is0_read != avs_is1_read))//Testing reads
////         if ((asi_sink0_data == 1'b0) || (avs_is0_read != avs_is1_read) ||
////         (avs_is0_read && (avs_is0_address != avs_is1_address)))//Combined the two
////         if (/*(asi_sink0_data == 1'b0) || */(avs_is0_read != avs_is1_read) ||
////         (avs_is0_read && (avs_is0_address != avs_is1_address)) ||
////         (avs_ds0_read != avs_ds1_read) || (avs_ds0_write != avs_ds1_write) ||
////         ((avs_ds0_read || avs_ds0_write) && (avs_ds0_address != avs_ds1_address))
////         || ((avs_ds0_read || avs_ds0_write) && (avs_ds0_byteenable != avs_ds1_byteenable))
////         || (avs_ds0_write && (avs_ds0_writedata != ds1_writedata)))//Full implementation of
the original error_detected
////            //code with the push button condition commented out to test the error injection
methods.
//        if(error_detected)//Same and the last design, but much easier to read.
//           begin
//              aso_source0_data = 1'b0;
////               reset_source <= 1'b1;//Active reset; disabled to test error injection
//              reset_source <= 1'b0;//SWITCH ME BACK!!!!
//           end
```

```
//        else
//           begin
////               aso_source0_data = 1'b1;
//              aso_source0_data = aso_source0_data;
//              reset_source <= 1'b0;
//           end
//     end
//end
//assign aso_source0_data = reset ? 1'b1 : (error_detected ? 1'b0 :
aso_source0_data);//Simplified LED controls.
//Since the reset was disabled the only thing the code above was doing was controlling the LED,
which this imbedded ternary statement does,
// without being clocked.
wire error_detected01;//Detects an error/difference between cpu_0 and cpu_1 slave input signals
assign error_detected01 = ((avs_is0_read != avs_is1_read) ||
                        (avs_is0_read && (avs_is0_address != avs_is1_address)) ||
                        (avs_ds0_read != avs_ds1_read) || (avs_ds0_write != avs_ds1_write) ||
                        ((avs_ds0_read || avs_ds0_write) && (avs_ds0_address != avs_ds1_address))
                        || ((avs_ds0_read || avs_ds0_write) && (avs_ds0_byteenable !=
avs_ds1_byteenable))
                        || (avs_ds0_write && (avs_ds0_writedata != avs_ds1_writedata)));

wire error_detected02;//Detects an error/difference between cpu_0 and cpu_2 slave input signals
assign error_detected02 = ((avs_is0_read != avs_is2_read) ||
                         (avs_is0_read && (avs_is0_address != avs_is2_address)) ||
                         (avs_ds0_read != avs_ds2_read) || (avs_ds0_write != avs_ds2_write) ||
                         ((avs_ds0_read || avs_ds0_write) && (avs_ds0_address !=
avs_ds2_address))
                         || ((avs_ds0_read || avs_ds0_write) && (avs_ds0_byteenable !=
avs_ds2_byteenable))
                         || (avs_ds0_write && (avs_ds0_writedata != avs_ds2_writedata)));

wire error_detected12;//Detects an error/difference between cpu_1 and cpu_2 slave input signals
assign error_detected12 = ((avs_is2_read != avs_is1_read) ||
                         (avs_is2_read && (avs_is2_address != avs_is1_address)) ||
                         (avs_ds2_read != avs_ds1_read) || (avs_ds2_write != avs_ds1_write) ||
                         ((avs_ds2_read || avs_ds2_write) && (avs_ds2_address !=
avs_ds1_address))
                         || ((avs_ds2_read || avs_ds2_write) && (avs_ds2_byteenable !=
avs_ds1_byteenable))
                         || (avs_ds2_write && (avs_ds2_writedata != avs_ds1_writedata)));

reg disabled0;//Indicates if cpu_0 is disabled
reg disabled1;//Indicates if cpu_1 is disabled
reg disabled2;//Indicates if cpu_2 is disabled

//Disable
always @ (posedge clk)
begin
   if (reset)//Resets the disables to allow the CPUs to run
      begin
         disabled0 = 1'b0;
         disabled1 = 1'b0;
         disabled2 = 1'b0;
      end
   else
      begin
         if(error_detected01||error_detected02||error_detected12)
            begin
               disabled0 = ((error_detected01 == 1'b1) &&
               (error_detected02 == 1'b1)) ? 1'b1 : disabled0;//Sets disable0 high if cpu_0
differs from cpu_1 and cpu_2
               disabled1 = ((error_detected01 == 1'b1) &&
               (error_detected12 == 1'b1)) ? 1'b1 : disabled1;//Sets disable0 high if cpu_1
differs from cpu_0 and cpu_2
               disabled2 = ((error_detected12 == 1'b1) &&
               (error_detected02 == 1'b1)) ? 1'b1 : disabled2;//Sets disable0 high if cpu_2
differs from cpu_0 and cpu_1
            end
         else
```

```
            begin//The system is one designed to work with three or two operational CPU, so if a
second CPU fails and is diasbled the third must be disabled as well
               disabled0 = ((disabled1 == 1'b1) && (disabled2 == 1'b1)) ? 1'b1 : disabled0;//If
cpu_1 and cpu_2 are disabled cpu_0 is disabled as well
               disabled1 = ((disabled0 == 1'b1) && (disabled2 == 1'b1)) ? 1'b1 : disabled1;//If
cpu_0 and cpu_2 are disabled cpu_1 is disabled as well
               disabled2 = ((disabled0 == 1'b1) && (disabled1 == 1'b1)) ? 1'b1 : disabled2;//If
cpu_0 and cpu_1 are disabled cpu_2 is disabled as well
            end
      end
end

//Avalon Memory Mapped Signal Control
always @ (*)//Depending on which CPUs are disabled, some Avalon signals need to be modified
begin
   avs_is0_waitrequest = (disabled0) ? 1'b1 : avm_im0_waitrequest;//Sets instruction waitrequest
active if cpu_0 needs to be disabled
   avs_is1_waitrequest = (disabled1) ? 1'b1 : avm_im0_waitrequest;//Sets instruction waitrequest
active if cpu_1 needs to be disabled
   avs_is2_waitrequest = (disabled2) ? 1'b1 : avm_im0_waitrequest;//Sets instruction waitrequest
active if cpu_2 needs to be disabled
   avs_ds0_waitrequest = (disabled0) ? 1'b1 : avm_dm0_waitrequest;//Sets data waitrequest active
if cpu_0 needs to be disabled
   avs_ds1_waitrequest = (disabled1) ? 1'b1 : avm_dm0_waitrequest;//Sets data waitrequest active
if cpu_1 needs to be disabled
   avs_ds2_waitrequest = (disabled2) ? 1'b1 : avm_dm0_waitrequest;//Sets data waitrequest active
if cpu_2 needs to be disabled
   avm_dm0_writedata = (disabled0) ?
   (avs_ds1_writedata | avs_ds2_writedata) : (disabled1) ?
   (avs_ds0_writedata | avs_ds2_writedata) : (disabled2) ?
   (avs_ds0_writedata | avs_ds1_writedata) :
   (avs_ds0_writedata | avs_ds1_writedata | avs_ds0_writedata);//ORs the active data writedata
signals
   avm_dm0_address = (disabled0) ?
   (avs_ds1_address | avs_ds2_address) : (disabled1) ?
   (avs_ds0_address | avs_ds2_address) : (disabled2) ?
   (avs_ds0_address | avs_ds1_address) :
   (avs_ds0_address | avs_ds1_address | avs_ds2_address);//ORs the active data address signals
   avm_dm0_write = (disabled0) ?
   (avs_ds1_write & avs_ds2_write) : (disabled1) ?
   (avs_ds0_write & avs_ds2_write) : (disabled2) ?
   (avs_ds0_write & avs_ds1_write) :
   (avs_ds0_write & avs_ds1_write & avs_ds2_write);//ANDs the active data write signals
   avm_dm0_read = (disabled0) ?
   (avs_ds1_read & avs_ds2_read) : (disabled1) ?
   (avs_ds0_read & avs_ds2_read) : (disabled2) ?
   (avs_ds0_read & avs_ds1_read) :
   (avs_ds0_read & avs_ds1_read & avs_ds2_read);//ANDs the active data read signals
   avm_dm0_byteenable = (disabled0) ?
   (avs_ds1_byteenable | avs_ds2_byteenable) : (disabled1) ?
   (avs_ds0_byteenable | avs_ds2_byteenable) : (disabled2) ?
   (avs_ds0_byteenable | avs_ds1_byteenable) :
   (avs_ds0_byteenable | avs_ds1_byteenable | avs_ds2_byteenable);//ORs the active data
byteenable signals
   avm_im0_address = (disabled0) ?
   (avs_is1_address | avs_is2_address) : (disabled1) ?
   (avs_is0_address | avs_is2_address) : (disabled2) ?
   (avs_is0_address | avs_is1_address) :
   (avs_is0_address | avs_is1_address | avs_is2_address);//ORs the active instruction address
signals
   avm_im0_read = (disabled0) ?
   (avs_is1_read & avs_is2_read) : (disabled1) ?
   (avs_is0_read & avs_is2_read) : (disabled2) ?
   (avs_is0_read & avs_is1_read) :
   (avs_is0_read & avs_is1_read & avs_is2_read);//ANDs the active instruction read signals
end

//Error LED
//Here are two equally viable LED control options.  Each behaves identically, and are completely
interchangeable.
```

```
//Both trigger the LED at the first sign of an error and keep the LED on until all errors are
gone, during reset.
//assign aso_source0_data = (disabled0||disabled1||disabled2) ? 1'b0 : 1'b1;
assign aso_source0_data = (error_detected01||error_detected02||error_detected12) ? 1'b0 : 1'b1;

//Reset Control
always @ (posedge clk)//Globally resets the system if all three CPUs are disabled
begin
   if (reset)
   begin
         reset_source <= 1'b0;
      end
   else
      begin
//       if(disabled0 && disabled1 && disabled2)//DO NOT USE IF HUMANS ARE MANUALLY INJECTING
ERRORS!! Please use the push button conditions below.
         if(disabled0 && disabled1 && disabled2 && asi_sink0_data && asi_sink1_data &&
asi_sink2_data)//Only resets when the push button are release,
         //that no errors remain through reset
//       if(disabled0 && disabled1 && disabled2 && button0 && button1 && button2)//Debounce
circuit design
            begin
               reset_source <= 1'b1;//Sends reset signal
            end
         else
            begin
               reset_source <= 1'b0;
            end
      end
end

//Debounce Circuit

//reg button0;
//reg button1;
//reg button2;
//
//reg [15:0] count0;
//reg [15:0] count1;
//reg [15:0] count2;
//
//always @ (posedge clk)
//begin
//  if (asi_sink0_data == 1'b0)
//  begin
//    count0 = count0 + 1;
//    if (count0 == 50000)
//    begin
//       button0 = 1'b0;
//    end
//    else
//    begin
//       button0 = 1'b1;
//    end
//  end
//  else
//  begin
//     count0 = 1'b0;
//     button0 = 1'b1;
//  end
//  if (asi_sink1_data == 1'b0)
//  begin
//     count1 = count1 + 1;
//     if (count1 == 50000)
//     begin
//        button1 = 1'b0;
//     end
//     else
//     begin
//        button1 = 1'b1;
//     end
```

```
//  end
//  else
//  begin
//     count1 = 1'b0;
//     button1 = 1'b1;
//  end
//  if (asi_sink2_data == 1'b0)
//  begin
//     count2 = count2 + 1;
//     if (count2 == 50000)
//     begin
//        button2 = 1'b0;
//     end
//     else
//     begin
//        button2 = 1'b1;
//     end
//  end
//  else
//  begin
//     count2 = 1'b0;
//     button2 = 1'b1;
//  end
//end

endmodule
```