Theses and Dissertations   1. Thesis and Dissertation Collection, all items

2006-12

Implementation of Configurable Fault Tolerant Processor (CFTP) experiments

Caldwell, Gerald W.

Monterey California. Naval Postgraduate School

https://hdl.handle.net/10945/2380

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

IMPLEMENTATION OF CONFIGURABLE FAULT TOLERANT PROCESSOR (CFTP) EXPERIMENTS

by

Gerald W. Caldwell

December 2006

Thesis Advisor:	Herschel H Loomis, Jr.
Second Reader:	Alan A. Ross

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

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

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE December 2006	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE: Implementation of Configurable Fault Tolerant Processor (CFTP) Experiments			5. FUNDING NUMBERS
6. AUTHOR(S) Gerald W. Caldwell			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING/MONITORING AGENCY REPORT NUMBER

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

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

13. ABSTRACT (maximum 200 words)

The Configurable Fault Tolerant Processor (CFTP) team at Naval Postgraduate School (NPS), Monterey, was created to develop, test, and implement reliable computing solutions for the space environment. The CFTP team seeks to design reliable circuits using Field Programmable Gate Arrays (FPGA) to include designs that mitigate the radiation hazards posed to FPGAs. A significant challenge faced by the CFTP team has been the integration and subsequent software development of the CFTP architecture, which includes a "Controller" and an "Experiment" FPGA.

This thesis investigates some of the specific design issues that must be considered for future experiments, to include timing between the two FPGAs, and data throughput of the CFTP architecture. Procedures for the development and implementation of experiments are detailed for the benefit of future experimenters who may be new to designing for FGPAs. Lastly, the Controller program is streamlined such that only minor modifications are required by prospective users in order to conform to specific experiments.

Over the years the CFTP team has produced several experiments that will provide reliable computing solutions for the space environment. Now, in addition to the "what" is to be used in space, this thesis presents "how" to run them in space.

14. SUBJECT TERMS Field Programmable Gate Array (FPGA), Single Event Upset (SEU)			15. NUMBER OF PAGES 129
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

**IMPLEMENTATION OF CONFIGURABLE FAULT TOLERANT
PROCESSOR (CFTP) EXPERIMENTS**

Gerald W. Caldwell
Major, United States Marine Corps
B.A., Emory & Henry College, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
December 2006**

Author: Gerald W. Caldwell

Approved by: Herschel H. Loomis, Jr.
                 Thesis Co-advisor

                 Alan A. Ross
                 Thesis Co-Advisor

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

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Configurable Fault Tolerant Processor (CFTP) team at Naval Postgraduate School (NPS), Monterey, was created to develop, test, and implement reliable computing solutions for the space environment. The CFTP team seeks to design reliable circuits using Field Programmable Gate Arrays (FPGA) to include designs that mitigate the radiation hazards posed to FPGAs. A significant challenge faced by the CFTP team has been the integration and subsequent software development of the CFTP architecture, which includes a "Controller" and an "Experiment" FPGA.

This thesis investigates some of the specific design issues that must be considered for future experiments, to include timing between the two FPGAs, and data throughput of the CFTP architecture. Procedures for the development and implementation of experiments are detailed for the benefit of future experimenters who may be new to designing for FGPAs. Lastly, the Controller program is streamlined such that only minor modifications are required by prospective users in order to conform to specific experiments.

Over the years the CFTP team has produced several experiments that will provide reliable computing solutions for the space environment. Now, in addition to the "what" is to be used in space, this thesis presents "how" to run them in space.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First and foremost, I would like to thank my wife, Laurie and my girls, Danielle and Sara, for their understanding during all the days I was home, but not really home, hiding in my office writing this thesis.

Special thanks are owed to the following people, without whom this thesis would not have been possible.

To Major Joshua Snodgrass, USAF, for his patience and valuable help with VHDL code, and hours of dedication to the CFTP team.

To Mindy Surratt for her help in my understanding of how the architecture works, and her dedication to the CFTP team and creating reference [1].

To Professors Loomis and Ross for their guidance, patience, and support.

To Ron Aikins for his dedication over several weekends to get the CFTP project into space.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The Configurable Fault Tolerant Processor (CFTP) project was created for the purpose of developing and testing fault tolerant circuits in space. Redundancy is one solution to the hazards that radiation in the space environment presents to electronic circuits. Field Programmable Gate Arrays (FPGA) provides a viable test bed for fault tolerant experiments due to their flexibility and ability to be programmed multiple times. The CFTP team created a robust architecture specifically designed to test and evaluate fault tolerant circuits through the use of FPGAs [4].

The primary components of this architecture are; two FPGAs, a PC/104 bus, an ARM processor, and Flash Memory. One FPGA is designated as the Controller FPGA, such that it controls the loading and running of experiments, as well as data transport over the PC/104 bus. The other FPGA is designed to be the Experiment FPGA, dedicated solely for the implementation of fault tolerant circuits.

Designers for the CFTP team over the years have created many interesting experiments that provide viable solutions for fault tolerant circuits. However, in addition to the design phase of an experiment, much effort has been spent understanding how a circuit on the Experiment FPGA interfaces with the Controller FPGA. Often times getting an experiment to properly integrate within the CFTP architecture proved much tougher than the design of the experiment itself. The original goal of this thesis was to provide future designers with the necessary insight into the inner workings of the Controller such that more effort can be directed towards designing experiments and less effort towards how they are implemented.

This thesis begins with an overview of the architecture and discussion of the code that is the design of the Controller. The purpose and functions of the Controller FPGA are discussed in detail to include clocking issues and how it interfaces with other components on the CFTP architecture. Emphasis is provided on portions of the Controller code that prospective designers will have to consider modifying to meet the needs of their experiments.

Beyond how the Controller FPGA, X1, interfaces with the Experiment FPGA, X2, this thesis explores timing and synchronization between the two chips. Several designs are implemented on both X1 and X2 showing that the two chips can be synchronized to run at the same clock rate, successfully transferring data without the use of handshaking signals. Also investigated is the maximum safe data rate that can be achieved across the PC/104 bus.

Finally, this thesis provides an example design that is implemented onto the CFTP architecture. This example design highlights the functionality of TMR while demonstrating how to account for many of the integration issues within the CFTP architecture. More importantly, the design demonstrated in Chapter V provides prospective designers a clear example of how the code within X1 is modified to suite the needs of an experiment implemented on X2.

The experimental design presented in Chapter V has been implemented on both the Flight and Development Boards, and its output is included toward the end of Chapter IV. This design has been installed on the Flight Board and will be the first experiment to provide output from the CFTP project shortly after its launch on 18 January 2007.

I. INTRODUCTION

Computing in the space environment is a challenge due to the inherent radiation environment and the subsequent adverse effects on electronic circuits. Additionally, long development schedules for space circuits have created a growing demand for increased flexibility. Field Programmable Gate Arrays (FPGAs) are one answer due to their inherent flexibility and their capability to be reconfigured. However, the radiation susceptibility of FPGAs can lead to data and configuration errors. This is most commonly caused by Single Event Upsets (SEU), where radiation causes logical bit values to change.

The Configurable Fault Tolerant Processor (CFTP) program at Naval Postgraduate School (NPS), Monterey, was initiated several years ago, and has evolved into a robust experimental platform [4]. Two separate architectures, both containing two Xilinx Virtex FPGA chips for implementing experiments, have been designed by the CFTP team and are fully functional. These architectures are structured to effectively enable the implementation of space-born experiments, and more importantly, to easily store and download the results for evaluation. The main components of the two architectures are as follows: Experiment FPGA, Control FPGA, PC/104 Bus interface, Flash Memory, EEPROM (Electronically Erasable & Programmable Memory), and an ARM (Advanced RISC Machine) Processor running an embedded Linux operating system [1].

The control FPGA is designed to be a controller for the loading of experiments, and passing data to the ARM processor through a PC/104 interface. The experimental FPGA is just that – an FPGA that is used to test various radiation-hardened designs that mitigate the effects of SEUs [4].

The CFTP team strives to design radiation hardened circuits that mitigate the effects SEUs have on FPGAs. This approach within the CFTP architecture presents other challenges beyond finding viable techniques for reliable computing, to include the integration among two FPGAs, a PC/104 bus, and an ARM processor. This thesis

explores those issues as well as the processes by which experiments are actually implemented. Specifically, this thesis will serve as a manual for future prospective experimenters on the CFTP team.

A. CFTP OBJECTIVE

The objective of the CFTP program at Naval Postgraduate School is to design reconfigurable and reliable space-based computer systems through the use of commercial-of-the-shelf (COTS) FPGAs. Because of the need for reliable electronic circuits in space, it is essential to have the ability to reconfigure and/or redesign space-born processors. FPGAs provide an ability to perform reconfigurations, and therefore offer great flexibility. The CFTP team seeks to design radiation hardened circuits through software solutions in order to counter one of the primary limitations of FPGAs – susceptibility to SEUs.

B. RESEARCH OBJECTIVES

This objective of this thesis is to detail the tools and techniques for implementing experiments, and to investigate timing constraints and integration issues on the CFTP architecture. The past years of development by students working on the CFTP team have created many lessons learned and produced many interesting designs. As a result, this thesis provides a formal document detailing the complicated and sometimes intricate procedures for developing and implementing experiments.

C. BACKGROUND

For the past several years numerous students on the CFTP team have concentrated on the mitigation of SEUs. The work up to this point has primarily focused on creating reliable computing solutions for the space environment using triple modular redundancy (TMR). Pete Majewicz created a processor for implementation on an FPGA that uses internal TMR, which he named the PIX processor [2]. James Coudeyras concentrated on a design that uses the entire FPGA chip to increase the probability of an SEU occurring, thereby enabling the testing of the process by which an SEU is detected and corrected [3].

Dean Ebert's thesis is the initial work that determined the design of the current CFTP architecture [4]. This work defined many of the issues considered in the initial design, and provided the solutions and final integration decisions that made up what the CFTP architecture is today.

D.   CFTP ENVIRONMENT

Within the CFTP development environment there are two separate architectures that provide two separate functions. One of the architectures is named the "Development Board," and its function is self-descriptive; to provide a platform for developing and testing experiments before implementation in space. The other architecture is named the "Flight Board," and is function is self-descriptive as well; a platform for running experiments in space. The basic architecture of the Development Board and the Flight Board is identical. Both contain two FPGAs and the interface and support components.

The primary difference between the Development Board and the Flight Board are the FPGAs themselves. The two FPGAs on the Flight Board are total-dose RADHARD (radiation hardened), and are therefore intended for flight in space. The Development Board uses two MILSPEC (military specification) FPGAs that are not designed to survive the space environment. The two types of FPGAs are mounted in two different types of packages, which means that their pinouts differ. Therefore, the FPGA design files and constraint files must be slightly different (see Appendix B for specific constraint file (UCF) considerations).

Though the development of an experiment can theoretically involve the creation of a circuit on one FPGA alone, this is not sufficient for evaluating experiments in space. A means to control the implementation of various experiments, as well as data collection, is essential for evaluation and analysis. This requires not only the integration between two FPGAs, but the other support items as well, such as Flash Memory, the PC/104 bus, and the ARM processor. The procedures for creating and implementing experiments that take these integration issues into account dictates that prospective experimenters become familiar with how the architecture is integrated, and the specific limitations that result from that integration.

The primary limitation of the CFTP architecture is the maximum data rate that can be achieved across the PC/104 bus. This limitation is bounded by the interaction between the PC/104 bus and the ARM processor, and more specifically, the ability of the Linux operating system on the ARM to perform reads on the PC/104 bus while keeping

other processes running in the background. This thesis specifically addresses these limitations, providing detailed guidance for future CFTP team experimenters.

E.   IMPLEMENTING EXPERIMENTS

Successfully designing an experiment that produces output on an FPGA is merely the first milestone that must be completed. This is usually accomplished via logic design and simulation with Computer Aided Design (CAD) software, tailored to the specific type of FPGA for which the experiment is to be implemented. The CFTP architecture design has some limitations, which will be addressed in this thesis, and those limitations must be taken into account when designing an experiment.

Once an experiment is successfully tested via simulation, which must include the creation of an accompanying constraint file, the controller code, code that runs the Control FPGA, must be modified to work with the experiment. Though the modifications may be few and relatively trivial, the controller code must be changed and compiled so that it will pass the proper number of bits of data, and will pass that data at an appropriate data rate. Also, the constraint file within the controller code must be modified to match the constraint file of the experiment.

F.   OVERVIEW

Chapter II of this thesis gives the reader a brief overview of the CFTP architecture; how it is organized and some of the specifics of the various components. Chapter III discusses in detail the design of X1, the Controller. It provides future CFTP designers the necessary details to understand how the Controller interfaces with the rest of the CFTP architecture, and more importantly, the portions of the Controller code that must be modified when creating an experiment. Chapter IV provides even more insight into the inner workings of the CFTP architecture as it addresses timing and some key limitations of the Flight and Development Board. Chapter V reviews the processes for implementing an experiment onto the CFTP architecture, from the first stages of a Hardware Description Language (HDL) description and/or schematic development, to implementation on an FPGA chip. Finally, Chapter VI provides conclusions and some suggestions for future work.

II. CFTP ARCHITECTURE

Though two architectures have been developed by the CFTP team, they are nearly identical in layout, and they are functionally identical. The two architectures, the Development Board and Flight Board, were designed to accomplish the missions their names imply; the Development Board is for the development and testing of experiments on the ground, and the Flight Board is designed to implement experiments in space. Figure 1 is representative of both the Development and Flight boards.
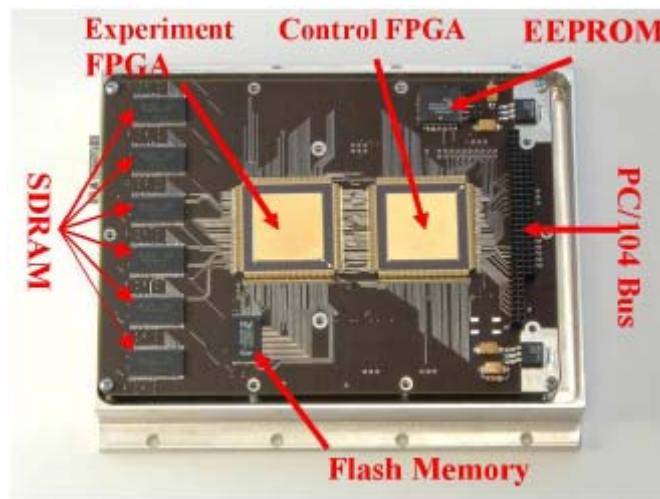
Figure 1.   CFTP Development Board (From Ref. [1]).

The two FPGA chips on the Flight Board are Xilinx Virtex I QPro Radiation Hardened FPGAs, with the specific Xilinx device number XQVR600 [6]. Table 1 shows the specifications for three Xilinx devices, showing where the XQVR600 falls with respect to other available devices. This highlights the number of programmable assets available within these FPGAs. Though the XQVR1000 would allow for larger and more complex designs, the XQVR600 was chosen due to architectural and costing constraints.

Device	System Gates	CLB Array	Logic Cells	Maximum Available I/O	Block RAM Bits	Max Select RAM Bits
XQVR300	322,970	32x48	6,912	316	65,536	98,304
XQVR600	661,111	48x72	15,552	316	98,304	221,184
XQVR1000	1,124,022	64x96	27,648	404	131,072	393,216

Table 1.   Xilinx RADHARD FPGA Gate Counts (From Ref. [6].)

The only difference between the Development Board and Flight Board, as previously mentioned in the introduction, are the FPGA chips; Development Board FPGAs are MILSPEC and the Flight Board FPGAs are RADHARD. This difference resulted in different physical pin layouts between the two FPGA chips located on each printed circuit board. On the Development Board, there exist 45 physical pin connections for the passing of data bits (one pin equals one bit) between the two FPGAs. However, on the Flight Board there are only 43 pins available for the passing of data bits, and the pin layout is slightly different.

The physical difference in the pin assignments were specifically for the SelectMap processes performed by the Controller FPGA. SelectMap is a hardware configuration mode that provides the fastest option for presenting data to an FPGA from a microprocessor [8]. Discussed in detail in Chapter III and Appendix B, the different pin numbers are assigned within the UCF files.

A. CONTROL FPGA

Also known as X1, this is the heart of the CFTP architecture. It provides the necessary interface for operations and data flow between the Experiment FPGA, the PC/104 Bus and the ARM processor. It controls the loading of an experiment's configuration from the Flash Memory to the Experiment FPGA. It is named the Controller FPGA because that is its overall function; to control the loading and running of experiments on the Experiment FPGA.

The Controller FPGA also has the responsibility to perform periodic scrubbing, (comparing the configuration stored in the Flash Memory with the configuration loaded on the Experiment FPGA) and can reconfigure the Experiment FPGA if warranted. The existence of a Controller FPGA provides a crucial capability for prospective experimenters; the ability to evaluate the reliability of space-born designs.

B. EXPERIMENT FPGA

Also known as X2, this is the FPGA for the implementation of specific fault-tolerant circuits. Designers can create a fault-tolerant circuit, implement it on the Experiment FPGA, and determine the ability of their design to function reliably as a standalone circuit. The existence of two FPGAs is central to the philosophy of the CFTP

architecture. In order to evaluate the reliability of a design, data produced from that design has to be evaluated. More specifically, the integrity of the design's configuration has to be monitored.

SEUs are a problem for FPGAs, not just because data might be altered, but because the configuration can become corrupt and change the *operation* of a circuit. It is this reason that the design of a circuit must implement some form of fault tolerance to mitigate, if not completely eliminate, the effects of SEUs in data and by detecting repeated faults caused by configuration errors so as to initiate configuration repair.

Note: RADHARD FPGAs are tolerant to large total dose radiation exposure, but are just as susceptible to SEUs as are non-RADHARD versions.

C. PC/104 BUS

The PC/104 bus is an 8-bit data bus, version 2.4, and is a trademark of the Embedded Consortium [5]. This is the interface between the Controller FPGA and the ARM processor. This is where data is transferred, and more importantly, it is the avenue through which the FPGAs are configured. A total of 104 signal contacts exist, though only 8 (8-bits) are for data transfers to and from an experiment. The other 96 pins are dedicated for functions such as handshaking between the PC/104 and X1, and other programming and loading processes.

D. EEPROM

This is a Xilinx component, XC18V04, and holds the initial configuration for X1, the Controller FPGA [4]. The purpose of this device is to configure X1 upon initial boot up for the CFTP architecture. The load within the EEPROM can not be changed once the Flight Board is attached to the satellite.

E. FLASH MEMORY

This device is an Intel Flash Configuration Memory (TE28F320C3), and is where the configurations for experiments to be implemented onto X2 are stored [4]. There is enough space in this memory module to store four separate configurations specifically for X2.

F. ARM PROCESSOR

The ARM processor is installed on a printed circuit board separate from the FPGAs, with a direct connection to the PC/104 bus. This is the interface between the

satellite and the two FPGAs, via the PC/104 bus. The ARM processor stores the programs that write and read to/from X1, and provides temporary storage of output data files, as well as the configurations for both X1 and X2.

The operating system on the ARM processor is an embedded Linux operating system that handles the various aforementioned processes. Various shell scripts and C-code programs have been developed to read data from the PC/104 bus, and to load configurations across the PC/104 bus for X1 and X2. One of the important processes the ARM must manage is a read program that performs constant polling to detect when data becomes available for reading on the PC/104 bus. Another important process the ARM manages is a write program that is invoked by the ARM to program X1 with its configuration file. The ARM uses this process to write data across the PC/104 bus and onto X1.

Without the ARM, it would not be possible to program X1 or X2, nor would it be possible to collect data from experiments on X2 and to pass that data to the satellite computer for eventual downlink to Earth. The two ARM processors on the Flight and Development Boards are identical and differ only in the number of processes they are required to run. These differences do not affect how experiments are designed, or how the code for X1 is implemented.

G.   SDRAM

This is a memory module available for the use of experiments on X2. Total random access memory (RAM) available is 16 megabytes (16 MB).

H.   CHAPTER SUMMARY

This chapter provided an overview of the organization of the CFTP architecture and its functionality. Very brief explanations of only the primary components were provided. The next chapter details the VHDL code that programs X1 into a controller. Also covered are considerations for future CFTP designers when developing code for programming an experiment onto X2.

III. MODIFYING CODE

VHDL (Very high speed integrated circuit Hardware Description Language) code describing the functions of the Controller, X1, has been developed and tested over the past years and is largely reliable. This VHDL code provides specific functions, all of which will be covered in this chapter. An understanding of the functionality of all the VHDL modules is essential as prospective experimenters are required to make minor modifications to three of the six modules that make up the VHDL code for X1 so that it will properly interface with their experiments.

The primary purpose of the Controller is to control the implementation and evaluation of experiments on X2, and it is arguably the most important component of the CFTP architecture. Without proper operation of X1, data from the experiments can not be collected. Although a designer should fully develop and test an experiment before making any modifications to X1, the functionality and required modifications to its VHDL code are covered first because of its importance. Section B provides some important considerations for the development of experiments for X2.

Beyond its purpose mentioned above, the Controller can also be modified to provide data to a circuit implemented on X2. This requires the designer to create and implement a new module, or a new process within an existing module, within the Controller code for X1. Chapter 4 provides methods for accomplishing this, as well as other scenarios describing how the Controller code can be modified to aid in the evaluation of circuits implemented on X2. This chapter limits its scope to the code that defines the Controller.

A.   CONTROLLER

The required capabilities of the Controller go beyond the simple functionality of a "pipe" for data transfer. Not only does the Controller transfer data, it controls the rate at which that data is transferred. More importantly, the Controller is designed to configure X2, the Experiment FPGA, as well as to perform periodic scrubbing and reconfiguration of X2 [4].

The VHDL code for the Controller is separated into six modules, as shown in Figure 2. These modules are; the Top Level module, which instantiates the other five modules and provides specific signal assignments, the X2 Interface, which contains the processes by which the two chips interface, the PC/104 Interface module, which provides the processes for data transfer across the PC/104 bus, two SelectMap modules, which provide processes for the loading and comparing of configurations, and a Clock Generator module, which performs clock division.
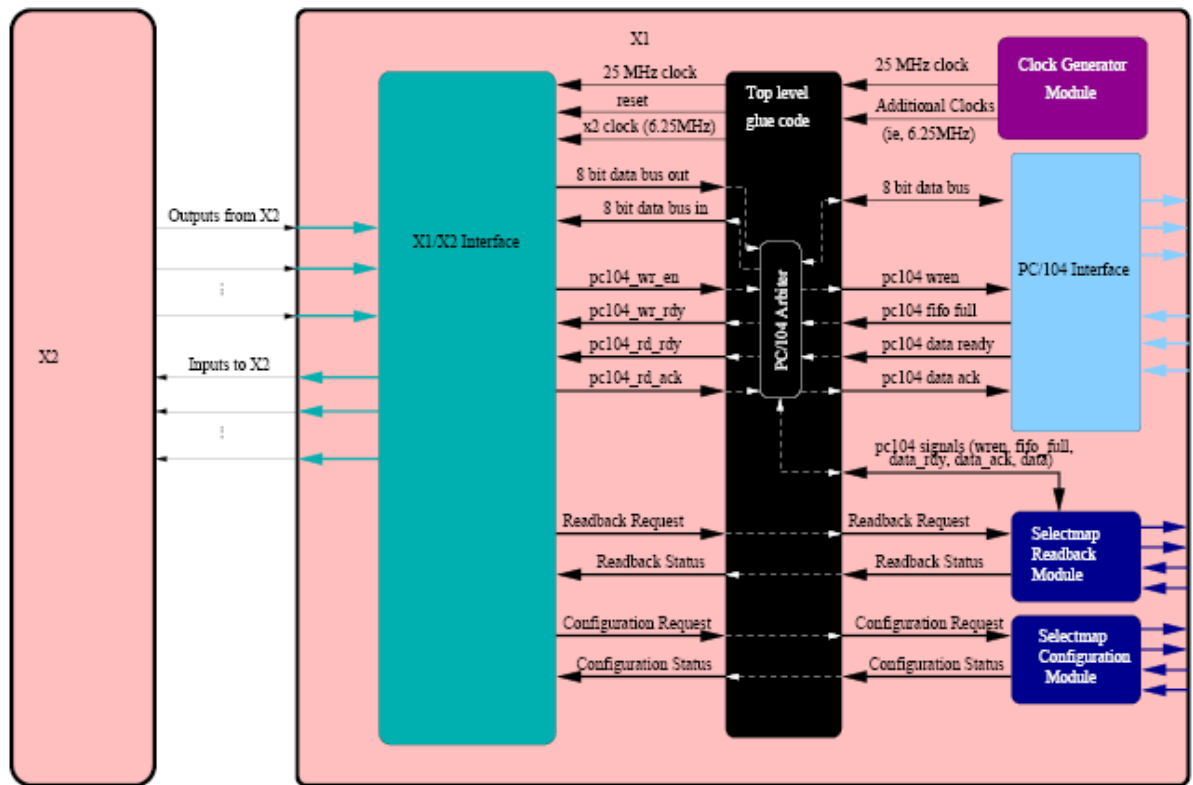
Figure 2.   Graphical Depiction of X1 Modules with X2 (From Ref. [1].)

1.   Top Level

As mentioned, aside from instantiation of the other five modules, the primary purpose of the top level code, named top_level.vhd, is simply signal assignment for the various modules. Prospective designers will only need to modify signals to suite the needs of an experiment within three sections in the top level code. Appendix B shows the specific sections of code containing the signals to be modified, with approximate line numbers denoting where the sections are located, within top_level.vhd. As is noted again

in subsection 2 for emphasis, these signal names exactly match the signal names in the port section of the X2 interface. Chapter V provides an example experiment which shows how these signals are named.

An important signal within the top level code that designers will not alter is the specific clock signal assignment. It is vital to have a full understanding of this clocking signal and how it can potentially affect design decisions for experiments.

The primary oscillator for the Flight and Development Boards comes from the ARM processor, and runs at 51 megahertz (MHz). This speed presented an early engineering dilemma for the CFTP team. Xilinx's application note 138 titled "Virtex FPGA Series Configuration and Readback" states that the SelectMap process can not perform simultaneous configurations of two devices at speeds equal to or greater than 50 MHz [7]. As a result of this constraint, the design decision was ultimately made to run all modules on the Controller at half the speed of the primary oscillator, using clock division performed via the clock generator module, discussed in subsection 7. The decision to clock-divide the primary oscillator by two, rather than fractional division to provide greater speed, was made in favor of simplicity. All of the Controller's VHDL modules run off this clock-divided signal, with the exception of initial clock signal coming into the top level module, because operating some modules at 51 MHz with the SelectMap modules at 25.5 MHz can lead significant timing problems.

For most situations, the use of this clock signal on X1 does not affect the speed at which experiments operate on X2. Designers can still use the primary oscillator and operate their circuits at 51 MHz. Chapter 4 shows how the timing between the two FPGAs shape some of the design decisions for experiments, proving that circuits on X2 can run at 51 MHz, and also providing a situation when X2 should be clock-divided to the same rate as X1.

2. **X2 Interface**

This VHDL module is the workhorse for X1, and it requires the greatest amount of modification by the designer. Within this module, x2Int.vhd, the designer will determine the byte size of the data stream, as well as the data rate across the PC/104 bus.

This module also determines how often a SelectMap read-back occurs, and when a SelectMap reconfiguration should take place.

The first consideration of any designer when modifying X1 should be the naming of signals to properly describe what type of data the experiment produces. It is worth emphasizing that any signal names modified within the primary port section of X1 must also be modified within three sections of the file "top_level.vhd." A complete listing of the X2 interface code is located in Appendix B for reference.

Once the proper signal-naming is complete, the designer next should consider what level of handshaking is required between the two FPGAs, if any at all. Because the two chips are synchronous, (their clocking signals are derived from the same 51 MHz oscillator), specific handshaking is not required, as is shown in Chapter 4, to transfer data from X2 across X1 and onto the PC/104 bus. The most common handshaking signal employed is an error-occurrence signal. This allows X1 to read data from X2 and write it to the PC/104 bus only upon the occurrence of a specific event. It is left up to the designer when the circuit on X2 should assert this specific signal high – perhaps upon the occurrence of a data error, or perhaps when the circuit finishes a calculation.

The module x2Int.vhd receives a reset signal from the top level code, named "RESET_i" in the x2Int.vhd port signal names, which resets all signals and vectors to predetermined values, defined within the behavioral of x2Int.vhd, when this signal goes high. The same purpose for X2 is served with the signal name "DATA_TO_X2_RESET_o." This signal is the same reset signal mentioned above, and both are derived from X1's clock. Therefore this is a synchronous signal, and sending it to X2 to start X2's processes in the same manner it is used to start the processes within X1, ensures that circuits running on both FPGAs are synchronized. Chapter IV details the importance of implementing this simple programming procedure.

Next the data size, the number of bytes to be transferred per write cycle, must be modified within x2Int.vhd to match the requirements of the experiment in X2. There are two variables with the X2 interface module which determine the data rate across the PC/104 bus. One variable determines the data size, and the other variable determines the

12

sampling rate. The sampling rate is defined as such; the periodicity at which X1 reads data from X2 and writes that data to the PC/104 bus.

The data rate, (refer to Chapter IV, "Timing," for the maximum safe data rate), is determined by multiplying the *sampling rate* times the *data size*. The sampling rate is set by changing the integer value assigned to a constant signal within x2Int.vhd, as below.

CONSTANT ERR_RPT_TIME   : integer := 38250000;

A process within x2Int.vhd uses this number as the final value of an internal counter. The counter increments on the 25.5 MHz clock, and when the count equals the constant ERR_RPT_TIME, the X2 interface module reads the appropriate signals for any data produced by X2, and the count resets to zero. The sampling rate is therefore determined as follows:

$$\frac{25.5\text{MHz}}{\text{ERR\_RPT\_TIME}} = \text{Sample Rate}$$

Also, when the count equals "ERR_RPT_TIME," a vector of bytes, the length of which is determined by the designer, is written to X1's output signals. The number of bytes written at each sample is determined by setting an integer value assigned to a constant signal.

CONSTANT REPORT_OUT_LENGTH : integer := 15;

The last two considerations for the experimenter are how often to perform a SelectMap read-back of the configuration of X2, and when a reconfiguration should take place. The standard within the CFTP development environment has been to perform a read-back of X2's configuration and compare it to the contents of the flash every 30 seconds. This timing is also determined through the use of a counter operating on the 25.5 MHz clock signal. The final value of this counter is another constant signal that can be modified by the designer if needed.

CONSTANT DLY_TIME : integer := 765000000;

Dividing this integer into 25.5 MHz yields the resulting timing for a read-back. In this case, 25,500,000 divided by 765,000,000 yields a rate of 0.0333 Hz. This translates to a read-back every 30 seconds.

13

Performing read-backs and comparing configurations every 30 seconds has not proven detrimental to the operations of X1 or X2. It is a relatively quick process, it does not interrupt operations of the circuit implemented on X2, and provides assurance that a configuration error will be caught and corrected in a timely fashion. To further guarantee the integrity of X2's configuration, the X2 interface can also initiate a reconfiguration if a certain number of data errors accumulate.

The number of data errors from the output of X2 is another threshold that can be set by the experimenter. The basic principle behind redundant computing allows for the occurrence and correction of data errors. Configuration errors on the other hand will cause repeated occurrence of the same data error, thus, repeated data errors are an indication of a potential configuration error. Therefore it must be decided when enough data errors have occurred such that the circuit should be reconfigured. The constant signal for that threshold within X2 is named "err_cnt," appropriately, and is a 24-bit standard logic vector. The specific value assigned to the threshold for this signal is located towards the end of the x2Int.vhd code, within an "IF" statement. The experiment implemented on X2 must define and calculate what this threshold should be; there is no definitive answer as experiments can vary greatly. However, the current practice within the CFTP project has been to set this value to hex 80.

3. Constraint File

The constraint file for X1, named control.ucf, is where signals are assigned to the specific pin locations on the X1 FPGA. All signal assignments within this file must exactly match the names of all signals within the port assignments in the top level code.

The two different architectures within the CFTP program, the Development and Flight boards, only differ within this file. As briefly mentioned on Chapter II, it is the physical pin assignments for the SelectMap processes that differ. Care must be taken by CFTP designers to ensure that the proper constraint file is being used for the Flight Board or Development Board. Though using the incorrect ucf-file is a mistake easily made, it is also just as simple to confirm that the correct ucf-file is being used. The top of each file is clearly commented on the top line as being designed for the Flight or Development Board, and deep into the files there exist comments denoting pin differences.

For the Development Board, two pins are available for assignment as the primary input clock for the top level code as there are two clocks available for use, a 50 and a 51 MHz clock. The example constraint file located in Appendix B points this out. For the Flight Board, the input clock assignment is straight forward as there is only one choice, the 51 MHz clock. For simplicity and compatability with the Flight Board, it is highly recommended that only the 51 MHz clock be used for the Development Board.

Lastly, the pin assignments between X1 and X2 for data flow must match in physical location, though the names themselves do not have to match exactly. A level of confusion can present itself here in that some of the corresponding signals are not the same pin numbers within the X2 and X1 constraint files. However, referring to the specific control.ucf code located in Appendix B, comments next to the respective line numbers within this file clearly denote how the signals correspond to one another. Signal names within X1's VHDL top level code must match the signal names in the X1 constraint file, and signal names within X2's VHDL top level code must match the signal names in the X2 constraint file.

4. PC/104 Interface

The sole purpose of this module, pc104IntArm.vhd, is to provide a means for interfacing between X1 and the PC/104 bus. To accomplish this, the PC/104 interface module employs a FIFO (first in, first out) buffer. This FIFO is 32-bits wide and 64-words deep. It was generated by CoreGen, an intellectual property of the Xilinx's Project Navigator Software. Experimenters are not required to make any modifications to this module.

A "maximum safe data rate" exists across the PC/104 bus and is defined as: the maximum rate at which output can be written to the PC/104 bus without any loss of data. This maximum safe data rate is a CFTP architecture limitation vice a specific limitation of the PC/104 bus itself. The interaction between the PC/104 Bus and the ARM processor, and specifically the processes running on the ARM, is what limits the data rate.

The FIFO employed with the PC/104 interface module is designed to stop accepting data if it becomes full. If the maximum rate at which the ARM can read is

exceeded, then data accumulates within the FIFO buffer. If data accumulates to the maximum size of the FIFO buffer, then it stops accumulating data until more space is available. This results in a loss of data. The procedures for determining the maximum safe data rate are described in Chapter IV.

The PC/104 bus is asynchronous therefore handshaking is employed to ensure proper data transfer. These handshaking signals, employed within the X2 interface module, identify when the PC/104 is being written to and therefore in a busy state, when it is ready to be written to, ready to be read from, as well as a signal that acknowledges a read. These four signals can be located in the port assignment section of "x2Int.vhd," and their respective functions are clearly commented.

5. SelectMap Configure

This module, SelectMap_config.vhd, performs the actual configuration of X2. When X1 is programmed with the Controller code described in this chapter, the first process executed is this module. The SelectMap configuration module reads the flash memory, starting at address zero, and takes the first 900 kilo-bytes (KB) of the flash memory and loads it into X2. It performs this process when commanded by the X2 interface module. Experimenters are not required to make any modifications to this module.

6. SelectMap Read Back

This module, SelectMap_readback.vhd, also performs as its name implies; it reads the configuration data loaded in the flash memory and compares it to the actual configuration loaded into X2. This process, known as scrubbing, is run periodically by the X2 interface module to ensure that the configuration in X2 has not been corrupted. This provides an extra layer of reliability to a fault-tolerant circuit programmed on X2. Experimenters are not required to make any modifications to this module, though they may wish to modify the interval at which scrubbing occurs.

7. Clock Generator

The only function of this module, "clockGen.vhd," is clock division of the primary oscillator, for reasons previously discussed. Two clock signals are generated from this module, one at 25.5 MHz and one at 3.1875 MHz. The specific assignment of the 25.5 MHz signal occurs with the top level module, "top_level.vhd," and is named

"s_clock." The 3.1875 MHz signal is provided as a convenience for designers who might require a slower clock. It is also assigned within "top_level.vhd" and is named "s_clock_x2." Designers are not required to make any modifications to this module.

B. EXPERIMENT

Prospective designers, when creating a circuit for the Experiment FPGA, have to first decide if the design will be created via CAD software, or if the design can be created via command-line editing in VHDL exclusively. If the design requires the use of schematics, then CAD software will be used. However, even if the design is exclusively created using VHDL, CAD software should be used unless the designer is an accomplished VHDL programmer. CAD software gives, in addition to a user friendly compiler, ready access to simulations which can be invaluable in verifying the proper operation of a circuit. The CFTP team has a license with the following software programs; 1) Xilinx's Project Navigator for project creations, compiling, place and route and mapping onto an FPGA. 2) ModelSim by Mentor Graphics for circuit simulation.

1. Implementation

Once a design has been successfully compiled and simulated, it must be implemented; the circuit has to be mapped, placed and routed onto the FPGA. All of the CLBs (configurable logic blocks) have to be configured to perform the desired operations of a prospective circuit. This is accomplished within Project Navigator with the command "Implement Design." If performed within Project Navigator, this will produce a file with an ".ncd" extension. This "experiment.ncd" file will need further modification, which is accomplished within the Linux operating environment on the CFTP server (see Appendix A for details). *It is vitally important that any circuit created for implementation onto X2 has a constraint file (experiment.ucf) added as a source to the respective project within Project Navigator BEFORE running "Implement Design."*

2. Flash File

The final file from any experiment to be implemented onto X2 is known as a flash file within the CFTP development environment, and has an ".fwr" file extension. This file is either created from the Project Navigator .ncd file or directly within Linux if command line editing of VHDL was used as the development process (See Appendix A). Whatever method is employed, this "experiment.fwr" file is what will be written to the

flash. It is the file that contains X2's final configuration. This is the file that X1 will use to program X2 and to verify the integrity of X2's configuration by using the SelectMap modules within X1.

3. Constraints

When creating a circuit for X2, there are a few constraints that the designer must consider for proper operation on either CFTP stack (Development or Flight). The number of pins available for data transfer between X1 and X2 vary slightly depending upon which specific architecture is used. If using the Flight Board, then 43 pins (43 bits) are available for data transfer between the two chips. However, if using the Development Board, then 45 pins (45 bits) are available. It is recommended that developers only use 43 pins when designing circuits for either stack as that is all that is available for space-born experiments. However, the extra two pins are available for the Development Board should a developer need those for specific trouble shooting.

Appendices A and B cover in detail how these pins are assigned. Developers must modify two constraint files (.ucf files); one for the Experiment design and one for the Controller design. Though the signal names within each file for each pin do not have to match by name, they must exactly match functionally. For example, output from X2 could be named "mult_out" for X2's constraint file, while that same signal could be named "input_from_X2" within X1's constraint file. These signals much match by pin number, and comments within the constraint files show how the pins on the two chips are connected by means of the circuit-board wiring.

A design policy within the CFTP project that experimenters should adhere to is: all signals traveling between X1 and X2, in the initial input or final output, must pass through clocked registers. In addition to being a policy within the CFTP project, using registers to collect data is an example of good programming as it ensures timing constraints are met which preserves the integrity of data. This requirement is discussed in Chapter IV.

C. CHAPTER SUMMARY

This chapter covered the inner-workings of the modules that comprise the circuit for X1, as well as some important considerations when developing code for experiments on X2. The next chapter provides a detailed analysis of timing within the CFTP architecture, and shows how the maximum safe data rate was determined.

THIS PAGE INTENTIONALLY LEFT BLANK

IV.   TIMING

Timing within the CFTP architecture, for both the Development and Flight Board, has required designers to carefully set values for counters that control the sampling rate within the Controller to ensure accurate data flow through the PC/104 bus. Even though no specific timing issues have been formally documented, it has been common practice within the CFTP team to keep the data rate low to prevent large accumulations of output data.

This chapter explores and subsequently demonstrates the maximum *safe* data rates that can be achieved across the PC/104 bus. As will be shown, this is not a limitation of the FPGA chips themselves. Data can be transferred between the two chips at the full rate of the oscillator. This maximum safe data rate limitation exists due to the interaction between the PC/104 bus and the ARM processor. This thesis defines the maximum safe data rate as: the maximum rate at which output from X2 can be transferred across the PC/104 bus without the loss of any data.

Though this chapter establishes a maximum safe data rate, it is also important to note that this data rate can change. The other processes running on the ARM limit the ability of the ARM to perform reads on the PC/104 bus. If processes within the ARM are added, removed, or altered, then the maximum safe data rate will change.

In general, experimenters have not been concerned with recording large volumes of sequential data produced by X2 at a high clock rate because most often the only important results are data that show the detection and/or correction of an SEU within a redundant circuit. Designers have generally verified results by one of three methods: comparing intermediate values derived from simulations, comparing a final result derived from several iterations (ignoring the intermediate values that lead to a final result), or comparing results by exception (output occurs only if triggered by a specific event).

This chapter demonstrates the procedures required to record sequential data produced from X2 and, in the process, determines the maximum safe data rate. This chapter also explores synchronization of the two FPGA chips, specifically addressing the

presence of clock skew between the two chips, showing through mathematical and empirical analysis that the clock skew is manageable at the full clock rate of 51 MHZ.

A.   CONTROLLER FUNCTIONS

Beyond the primary responsibilities of the Controller, there are four plausible scenarios for which the Controller can be programmed to facilitate the evaluation of experiments implemented on X2. All four of these scenarios are driven by timing – how the primary oscillator is used to drive the timing requirements of the circuits on X1 and X2. Of these four scenarios discussed in this section, three were implemented in support of experiments discussed in this chapter, and those results are included. The fourth scenario is left to future designers for potential implementation.

1.   Sampling Data

Circuits implemented on X2 can operate and produce data at the full rate, 51 MHz, of the primary oscillator. Because all of the modules on X1, except for the top level module, are clocked at 25.5 MHz, the data output from X2 must be sampled. This is demonstrated towards the end of Section C, and the methods for assigning the sampling rate have already been discussed in Chapter III.

2.   Clock Dividing X2

A circuit on X2 can be clock-divided for a myriad of reasons. One scenario that requires clock-division within X2 is when a designer wishes to view all data produced from an experiment in sequential order, rather than just a sampling of that data. To accomplish this, the designer slows the circuit on X2 down to the sampling rate on X1, such that X1 is reading at the same rate that the circuit on X2 is writing data. To do this, two levels of clock-division are required for the experiment; once down to the 25.5 MHz clock so that the two FPGAs run at the same rate, and the second division equal to the signal ERR_RPT_TIME discussed in Chapter III, so that the data can be passed through the ARM processor. This method is demonstrated in Section C.

3.   New Module for X1

Another method available to evaluate circuits is by creating a copy of the circuit on X2 and implementing it on X1 in a new module clocked at 51 MHz while the other preexisting modules on X1 remain at 25.5 MHz, and then using a voter clocked at 51 MHz to compare the outputs of the duplicate circuits. Then, if no errors are reported

from the voters on X2 and X1, the designer has increased assurance of correct data. This method still requires the final output of data across the PC/104 bus to be sampled as in method 1, but that sampling will contain two streams of data from two voters.

This method is employed in this chapter on a small scale using simple counters. A more interesting and complex example of this method can be reviewed in the Dissertation by Josh Snodgrass, where the Cordic algorithm is implemented on both X1 and X2 [10]

4.   Buffer on X1

Another design consideration that would allow all sequential data to be collected from a circuit on X2 running at 51 MHz is the implementation of a buffer on X1 that would temporarily store data. To implement this scenario, the circuit on X2 would need to be programmed to run for a certain time period then go into a wait state. The maximum safe data rate still can not be exceeded. The circuit on X2 would need to wait for the data within the buffer on X1 to be read before more data could be written. This method has not been explored by the author, although the FIFO discussed earlier provides this capability.

B.   DATA RATE

To determine the maximum safe data rate within the CFTP architecture, experiments were conducted in two phases. The first phase involved a simple design that was implemented on X1, temporarily removing X2 from the equation. This isolation of X1 simplified the design process and allowed for direct data collection through clock division. The second phase involves the implementation of the same experiment on X2, comparing the outputs of both X1 and X2 on both chips concurrently, and subsequently directing all outputs across the PC/104 bus. The second phase is important as it demonstrates that sequential data can be produced concurrently by both chips. In other words, phase two demonstrates that it is possible to synchronize both FPGAs such that they produce the same sequential data at the full rate of the CFTP oscillator, 51 MHz, without any handshaking. Phase two addresses the issues of clock skew.

1.   Phase One

A simple counter was implemented on the Controller, located within the counter that controls the sampling rate, and its sequential output recorded. Locating this counter

internal to the primary sampling rate counter within x2Int.vhd allowed for precise control over the rate at which the count is executed. A counter was used for this experiment because the output of a counter simplifies the verification of any disruptions in data flow; a number out of sequence is relatively easy to locate.

Referring to Table 2, the number of bytes outputted across the PC/104 bus remained constant throughout this phase, and the sampling rate, (defined in Chapter 3), was adjusted to achieve various data rates. For the purposes of this phase of the experiment, as noted in Table 2, the sampling rate served as the effective clock rate of X1 as no data was being produced from X2 for sampling. The signal ERR_RPT_TIME, (also defined in Chapter 3), was adjusted to achieve the data rates noted in Table 2. This signal was adjusted from a high sampling rate to a low sampling rate, incrementally, until no errors were detected in the output.

To summarize; this experiment was designed such that adjusting the signal ERR_RPT_TIME directly changes the speed of the count, as well as the rate at which data is written to the PC/104 bus.

As noted in the Observation section of Table 2, the higher data rates produce multiple, easy to recognize errors. For all the test runs, the errors themselves are the same in that there are large gaps of missing numbers, followed by a continuation of the count. The severity of these errors is directly related to the data rate; the higher data rates produce a greater number of errors and an earlier occurrence in the count sequence. The higher data rates also produce a greater gap between numbers before the count sequence continues in the output.

ERR_RPT_TIME	Effective Clock Rate	Bytes	Effective Data Rate (bytes/sec)	Observation
None	51 MHz	15	765 M	Multiple errors
5,000	10.2 KHz	15	153 K	Errors noted early in count
12,000	4.25 KHz	15	63.75 K	Errors noted later in the count
25,000	2.04 KHz	15	30.6 K	Less errors, occurring later
51,000	1.0 KHz	15	15 K	Less errors, occurring later
102,000	500 Hz (figure 3)	15	7.5 K	Error at number 0227
510,000	100 Hz (figure 4)	15	1.5 K	Error at number 0227
1,020,000	50 Hz	15	750	Error at number 0227
1,530,000	33.33 Hz (figure 5)	15	500	Perfect Data – no errors noted
2,550,000	20 Hz	15	300	Perfect Data – no errors noted

Table 2.   Data Rate Results from X1 Output.

These results are indicative of the FIFO buffer performing its job correctly. The FIFO buffer implemented within the PC/104 interface module, discussed in Chapter 3, is designed to stop accepting data when the capacity of the FIFO buffer is reached. At data rates that are well above the limits of the CFTP architecture, it is not unexpected that that the FIFO buffer will reach its capacity much faster than at rates that are closer to, but still above, the data rate limits.

Further inspection of the results in Table 2 reveals a pattern as the data rate was reduced closer to the limit; at 7500 bytes/sec (Bps) and lower, the first noted error began occurring at the same number. Referring to Figures 3, 4, and 5, the output of the count should be sequential. In other words, an error is defined as an interruption in the count sequence. Figure 5 provides two separate streams of output data with no errors, while Figures 3 and 4 show two streams of data with red circles denoting the locations of an error.

The red circle on the left data stream in Figure 3 denotes the first location of an error (interruption of sequential count) at number hex 0227. Referring to the left data stream in Figure 4 on the following page, a red circle again shows the first error occurring at hex 0227. Comparing the data within the red circles on the left in Figures 3 and 4, the amount of missing data after number 0227, the gap before the count sequence resumes, is greater in Figure 3 than in Figure 4. This clearly shows how the FIFO is able to recover quicker at slower data rates; the data in Figure 3 was output at a greater rate (7500 Bps) than the data in Figure 4 (1500 Bps). For the data in Figure 4, the FIFO recovered quicker and hence, the gap before the count sequence resumes is significantly shorter.
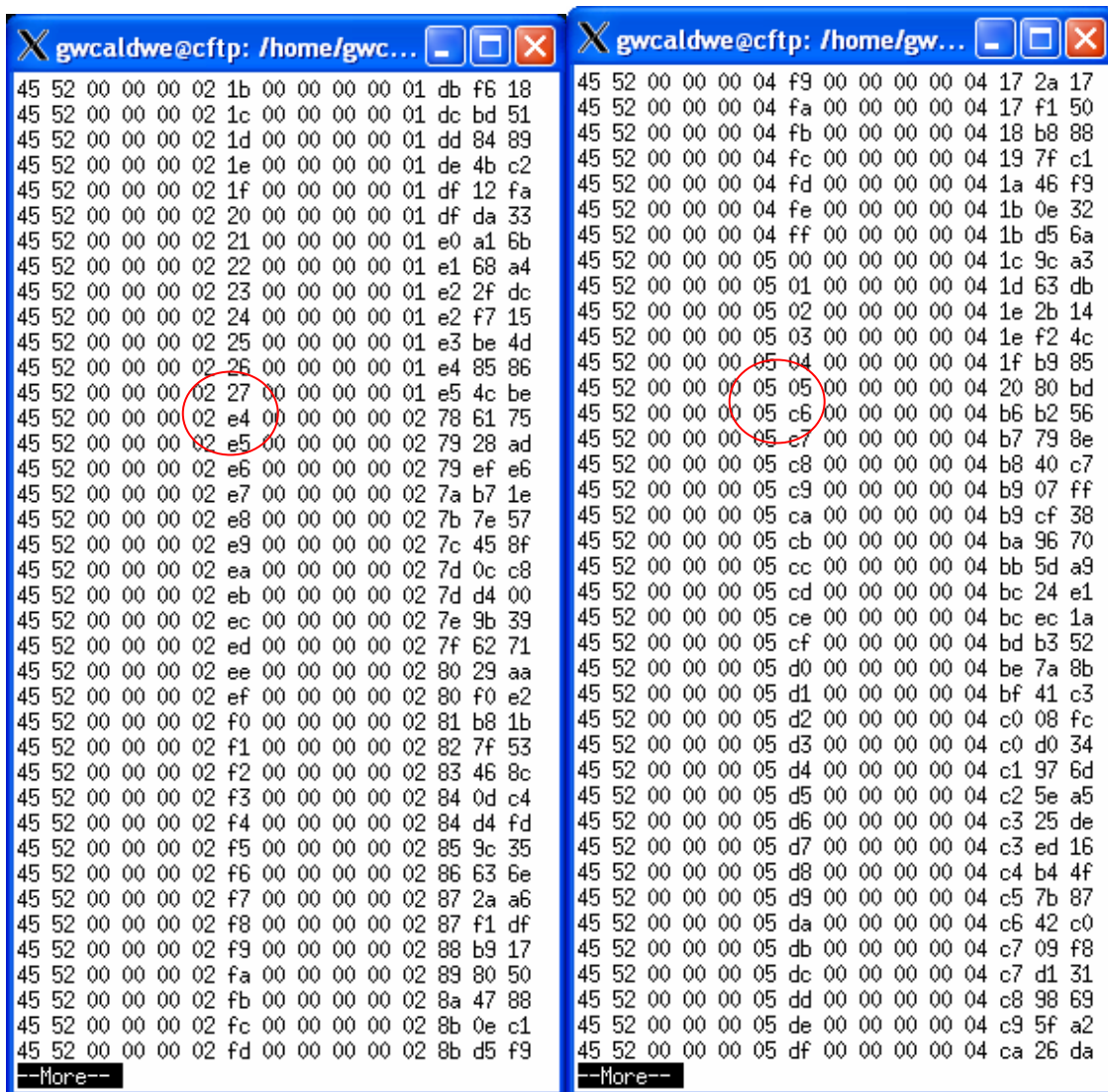
Figure 3.   Output from X1's Counter at 7500 Bytes/sec

Worth noting in Figures 3 and 4 are the data circled in red on the *right* side of the figures. At the higher data rate, (7500 Bps), the output of Figure 3 shows a second error sooner than the output in Figure 4. The lower data rate, (1500 Bps), for the data output of Figure 4 did not produce a second error until much later.

Figure 4.   Output from X1's Counter at 1500 Bytes/sec

Figure 5 shows output produced with no errors noted. Though this figure only shows the first sequence and the sequence with the common trend, (error at number 0227, the output was run for several minutes producing thousands of lines of data, and no errors were noted. Though this does not present irrefutable evidence that 500 Bps is the maximum safe data rate for the CFTP architecture, it is basis enough to conclude that the

noted data rate is slow enough for accurate data flow. More importantly, designers should not exceed this data rate if assurance is desired that no data has been lost, (FIFO stops accepting data until its size is reduced), before being written to the PC/104 bus.
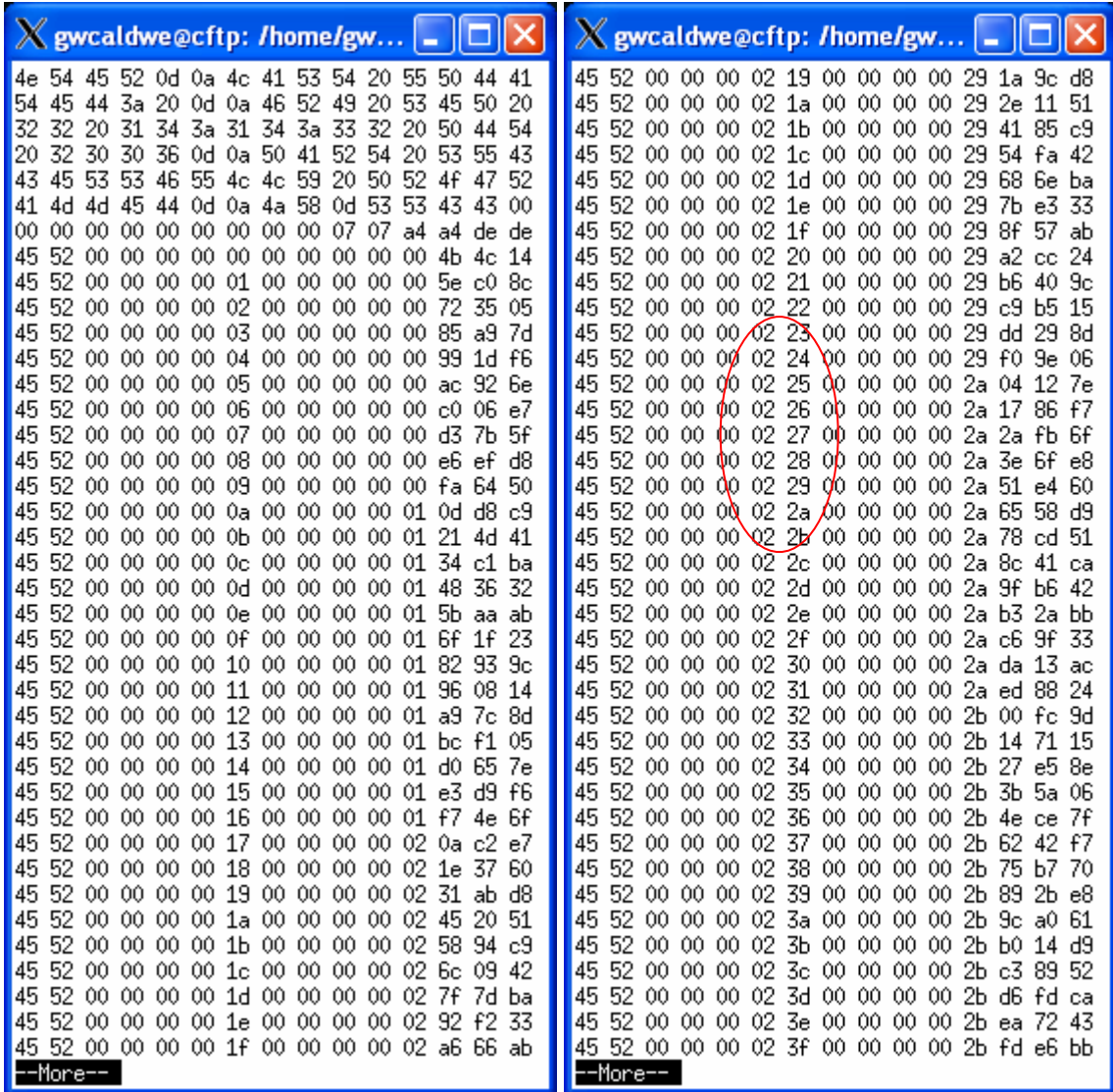

Figure 5.   Output from X1's Counter at 500 Bytes/sec

Though a maximum safe data rate of 500 Bps might initially appear to be a significant limitation, it is actually an acceptable parameter for the CFTP architecture. Because of memory limitations of the satellite platform, and available bandwidth with the uplink and downlink to the satellite, the amount of output data that can be collected at any one time is limited. The data rate of 500 Bps is a limitation that must be considered by designers if assurance of data integrity is desired when developing experiments for the

CFTP architecture. However, the actual output collection rate might need to be slower due to other limitations in space, or differences among the demands of the ARM processor on the Flight Board or Development Board.

2. Phase Two

For phase two, a counter was created for implementation on X2 with a voter to compare the results of the counter from X1 with the counter on X2. This was also done on X1; a voter added to compare X2's counter output with the counter on X1. To accomplish this, X1's counter output was directed to X2, and X2's counter output was directed to X1. The voters on each circuit are identical; they compare the two counter outputs and report the number three if the counters differ, else the voters produce the number zero if the counter outputs agree.

To synchronize the two voters on X1 and X2 only one signal needs to be passed from X1 to X2 – the reset signal discussed in Chapter III. For this dual-counter experiment, the reset signal generated by X1's top level VHDL module is passed to X2. This signal is also incorporated into the voter circuit on X2, initiating the count. The same thing was already written into the code for the counter on X1. This allows for synchronization of the two circuits upon initial startup.

In order to achieve a data rate at the rate of the oscillator between the two FPGAs, the SelectMap processes were disabled within x2int.vhd. This allowed the clock that runs the X2 interface module, located on X1, to be the 51 MHz oscillator instead of the 25.5 MHz signal coming from the Clock Generator module, which is required for the SelectMap processes as mentioned on Chapter III.

X1 can and normally does run off of two clocks, the primary oscillator signal, and the 25.5 MHz signal generated by the Clock Generator module. Therefore, in normal operations, X2 can produce data at 51 MHz and X1 can read that data at 51 MHz, and then sample it and send it across the PC/104 bus as previously discussed. For the purposes of this experiment only, all processes on X1 were clocked at 51 MHz to demonstrate that X1 and X2 can be synchronized without handshaking.

Figure 6 shows the first output sequence achieved with a data transfer at 51 MHz between the two FPGA chips. Even though the two counters are producing a count at 51

MHz, and the two voters are comparing those counts at the same rate, the output is only sampled because of the maximum safe data rate. Specifically, the output shows the following, in order:

1) An error-counter located in X1 that will increment only if the voter in X1 or the voter in X2 reports a value other than "00." This value will remain at "00" unless one of the voters reports an error. This error-counter is standard within X1 and is specifically addressed in Appendix B.

2) The results from the Voter located in X1 – this will report "00" if both counter outputs from X1 and X2 agree, otherwise it will report "03" if the counts differ.

3) The results from the Voter located in X2 – this will report "00" if both counter outputs from X1 and X2 agree, otherwise it will report "03" if the counts differ.

4) The count from the Counter located in X1.

5) The count from the Counter located in X2.

6) A timestamp generated in the Top Level code within X1.

```
X gwcaldwe@cftp: /home/gwcaldwe/results/dual_count                          _ □ X

JERRY'S DUAL COUNTER
LAST UPDATED:
WED SEP 27 09:20:37 PDT 2006
PART SUCCESSFULLY PROGRAMMED

timestamp: 00079c6b
Selectmap Reconfig...

Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 000a    Counter in X2: 000a    Timestamp: 00859de7
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0016    Counter in X2: 0016    Timestamp: 00d36fc8
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0022    Counter in X2: 0022    Timestamp: 012141a9
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 002e    Counter in X2: 002e    Timestamp: 016f138a
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 003a    Counter in X2: 003a    Timestamp: 01bce56b
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0046    Counter in X2: 0046    Timestamp: 020ab74c
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0052    Counter in X2: 0052    Timestamp: 0258892d
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 005e    Counter in X2: 005e    Timestamp: 02a65b0e
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0005    Counter in X2: 0005    Timestamp: 02f42cef
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0011    Counter in X2: 0011    Timestamp: 0341fed0
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 001d    Counter in X2: 001d    Timestamp: 038fd0b1
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0029    Counter in X2: 0029    Timestamp: 03dda292
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0035    Counter in X2: 0035    Timestamp: 042b7473
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0041    Counter in X2: 0041    Timestamp: 04794654
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 004d    Counter in X2: 004d    Timestamp: 04c71835
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0059    Counter in X2: 0059    Timestamp: 0514ea16
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0000    Counter in X2: 0000    Timestamp: 0562bbf7
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 000c    Counter in X2: 000c    Timestamp: 05b08dd8
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0018    Counter in X2: 0018    Timestamp: 05fe5fb9
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0024    Counter in X2: 0024    Timestamp: 064c319a
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0030    Counter in X2: 0030    Timestamp: 069a037b
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 003c    Counter in X2: 003c    Timestamp: 06e7d55c
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0048    Counter in X2: 0048    Timestamp: 0735a73d
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0054    Counter in X2: 0054    Timestamp: 0783791e
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0060    Counter in X2: 0060    Timestamp: 07d14aff
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0007    Counter in X2: 0007    Timestamp: 081f1ce0
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0013    Counter in X2: 0013    Timestamp: 086ceec1
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 001f    Counter in X2: 001f    Timestamp: 08bac0a2
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 002b    Counter in X2: 002b    Timestamp: 09089283
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0037    Counter in X2: 0037    Timestamp: 09566464
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0043    Counter in X2: 0043    Timestamp: 09a43645
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 004f    Counter in X2: 004f    Timestamp: 09f20826
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 005b    Counter in X2: 005b    Timestamp: 0a3fda07
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0002    Counter in X2: 0002    Timestamp: 0a8dabe8
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 000e    Counter in X2: 000e    Timestamp: 0adb7dc9
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 001a    Counter in X2: 001a    Timestamp: 0b294faa
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0026    Counter in X2: 0026    Timestamp: 0b77218b
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 0032    Counter in X2: 0032    Timestamp: 0bc4f36c
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 003e    Counter in X2: 003e    Timestamp: 0c12c54d
Error Count: 00    Voter in X1: 00    Voter in X2: 00    Counter in X1: 004a    Counter in X2: 004a    Timestamp: 0c60972e
--More--
```

Figure 6.   Dual Counter Output at 170 Bytes/sec (sampling rate of 10 Hz)

The output in Figure 6 clearly shows that data can be transferred at high rates across the two FPGA chips. It also demonstrates that data produced from X2 can be verified at a much lower rate – sampled below the rate at which it is produced. The data rate for the data produced in Figure 6 was set to 170 Bytes/sec by setting the signal ERR_RPT_TIME to 5,100,000. This establishes the sampling rate to be 10 Hz. The total number of bytes transferred per sample was 17; therefore the data rate produced is 170 Bytes/sec. However, the fact that no errors were detected verifies that the counters were running and transferring data between the two chips at the full rate of 51 MHz.

The data in Figure 6 demonstrates that two circuits, located separately in each FPGA chip, can be synchronized to run at the same rate. This synchronization phase of

31

the experiment was important for two reasons; one, it shows that data can be transferred between the two chips at the full rate of the oscillator, and two, it shows that clock skew is manageable, as discussed in section 3, below.

Figure 7.   Dual Counter Output at 173,400 Bytes/sec (sampling rate of 10.2 KHz)

The data in Figure 7 provides more evidence why designers should keep the sampling rate within X1 low enough such that the data rate is below 500 Bps. The data in Figure 7 was output across the PC/104 at 173,400 Bps – well above the maximum safe data rate. Referring to the area circled in red, it is clear that some data is missing. Closely inspecting the count from the top portion of this figure down to this area in red, it is clear that the sampling rate set in X1 produced a number on every third count. However, the area in red shows a gap much larger than three counts. This is evidence

that the FIFO buffer in the PC/104 interface module reached its capacity and stopped accepting data for a period of time. This gap of data is lost, and in this scenario, a designer would not know what this data might have shown.

Figure 8.   Dual Counter Output at 867 MBytes/sec.

Figure 8 shows data output at the full rate of the oscillator, 51 MHz, which is a data rate in this case of 867 Megabytes/sec. This output illustrates the danger of programming X1 to write to the PC/104 bus in excess of the maximum safe data rate. The output in Figure 8 appears to be good output – the counts match and the voters do not report any errors. However, as has been shown from the previous test results, data is missing from the output in Figure 8, even though it is not evident. It has been demonstrated that at rates well above 500 Bps, the maximum data rate of the PC/104 Bus is exceeded and the FIFO Buffer stops accepting data when at capacity. Though the data in Figure 8 is accurate, there are in fact thousands of bytes of missing data, which could be crucial to the evaluation of the reliability of a circuit. If there were errors in this missing data, then the error counter, (left column in Figure 8), would produce a number, but the data would not be available for analysis.

33

3. Clock Skew

Because the two FPGA chips on the CFTP architecture are run by the same oscillator, the potential exists for clock-skew caused errors. Determining the precise clock skew over the short distance between the two chips is a non-trivial matter, though certainly not impossible. However, if it can be shown that the clock skew is manageable, in other words, that the clock period inequality, Equation 1, is not violated, then the precise clock skew can be ignored for the purposes of experimental development. This section shows, via mathematical and empirical analysis, that this is indeed the case, that any clock skew present is not large enough to be of concern when developing an experiment for implementation onto the CFTP architecture. For mathematical analysis, the meta-stability equation was used, which is as follows:

$$T \geq 2S + t_{pdR} + t_{pdLogic} + t_{pdIB} + t_{pdOB} + t_{pdWiring} + t_s$$

In the meta-stability equation, also know as the clock period inequality, T stands for the clock period, which at 51 MHz is approximately 19.6 nanoseconds (ns), and S stands for the clock skew, which is not known. The purpose for multiplying clock skew times 2 is explained later. For the remaining terms, t_{pdR} is the flip-flop gate delay, $t_{pdLogic}$ is the logic delay, t_{pdIB} is the delay of input buffers and t_{pdOB} is the delay of output buffers, $t_{pdWiring}$ represents path delay for wiring, and finally, t_s is the flip-flop setup time.

As mentioned at the end of Chapter III, it is policy within the CFTP project that signals passed between X1 and X2 must pass through clocked registers. With this policy in mind, Figure 9 depicts how the terms within the clock period inequality apply to a signal passing from X2 to X1 regardless of how complex a circuit design might be on either chip. Figure 9 is not to scale as the two D flip-flops are significantly enlarged for clarity. The red arrows depict the path of the primary clock signal, and the data in the scenario depicted in Figure 9 is flowing from the D flip-flop on X2 to the D flip-flop on X1. This scenario would be precisely the same for data flowing in the opposite direction; the labels for X1 and X2 could merely be swapped.
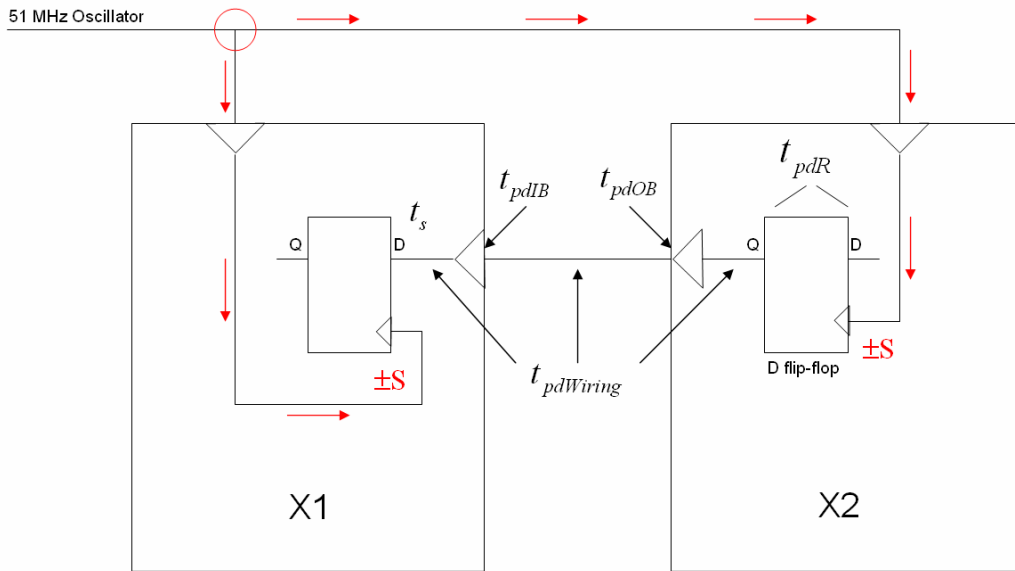
Figure 9.   Clock Skew Signal Paths

Values for the terms in the clock period inequality were determined via a synthesis report generated by the Xilinx compiler on the CFTP server, as well as within Xilinx's Project Navigator [10]. Two synthesis reports for two separate designs on X2 were reviewed, as well as a synthesis report for the typical circuit on X1, and the values for the terms were nearly identical for all the designs. One of those reports was for the dual counter experiment in the phase 2 discussion in Section B of this chapter.

Referring to Figure 9, the relation of each term from Equation 1 to a signal passing between X2 and X1 is depicted. Note the omission of the logic delay, $t_{pdLogic}$, because all signals passing between X1 and X2 are the outputs of registers and only wires exist between the two chips. The gate delay, t_{pdR}, is 1.372 nanoseconds (ns). The input buffer delay, t_{pdIB}, is 2.53ns and the output buffer delay, t_{pdOB}, is 5.672ns. The wiring delay between the buffers and the flip-flips, $t_{pdWiring}$, is 0.057ns, but the wiring delay between the two chips is an unknown quantity.

35

The setup time is defined as the time during which data input to a latch or flip must remain stable in order to guarantee the latched data is correct. The synthesis report does not provide a specific value called setup time. However, it does provide the delay of the signal at the D-input of the D flip-flop on X1, which is 0.84ns. If you include the delay of this D-input to the flip-flop, it appears in the inequality just like the setup time. Thus, it was concluded that the setup time for the FPGA flip-flops is 0.84ns.

As noted earlier, the skew is multiplied by 2 in the inequality. This is done to account for a worst case scenario. Referring to Figure 9, the path of the signal from the intersection circled in red to the input of a flip-flip represents the skew. Multiplying this value by 2 ensures that the longer of those two paths is taken into account.

Substituting the known values in Equation 1 and solving for S will yield a value for the allowable clock skew. However, there is still one unknown that must be accounted for before doing so; the wiring delay between the two chips. This unknown value can be conservatively estimated.

The distance between the two FPGAs is within a few millimeters, and the propagation of signals along wires is often calculated at the speed of light. To remain conservative, a distance of one centimeter is used, and one-half of the speed of light. This yields the following propagation delay between the two chips:

$$\text{Travel Time} = 0.01\text{m}/1.5*10^8\,\text{m/s} = 0.0667\text{ns}$$

Adding the values for each term, and omitting the logic delay and using three values for the wiring delay, the inequality is evaluated as follows:

$$19.6\text{ns} \geq 2\text{S} + 1.372\text{ns} + 2.53\text{ns} + 5.672\text{ns} + (0.057\text{ns} + 0.057\text{ns} + 0.0667\text{ns}) + 0.84\text{ns}$$

Adding the known terms on the right side, subtracting them from the clock period on the left side, and then dividing by two, yields an allowable clock skew of 4.503ns.

$$\text{S} \leq 4.503\text{ns}$$

Referring again to Figure 9, the travel time of the clocking signal from the intersection circled in red to the input of either flip-flop is likely to be quicker than 4ns.

To demonstrate this with another conservative calculation, a distance of ten centimeters is used this time with the same speed.

$$\text{Travel Time} = 0.1\text{m}/1.5*10^8 \text{ m/s} = 0.667\text{ns}$$

The signal from the intersection in Figure 9 must also pass through an input buffer, the delay of which was already determined to be 2.53ns, and the internal wiring delay was already determined to be 0.057ns. Adding the three terms together yields a value of 3.254ns.

As can be seen, even when using overly conservative values, it is likely that any skew between the two chips is not large enough to violate the clock period inequality. Further, phase 2 in Section B of this chapter provides empirical evidence that the inequality is not being violated.

Using these two forms of analysis (mathematical and empirical) it is clear that clock skew between the two FPGAs is manageable and does not need to be accounted for when designing experiments for the CFTP architecture, provided that signals that pass between the two FPGAs are the outputs of registers and go directly to register inputs. Specifically, it was shown that X2 can write data at 51 MHz, and X1 can read that same data at 51 MHz without any specific handshaking signals denoting the availability of that data. The only requirement for this to happen is the synchronization of the two chips with a reset signal generated by X1's top level module upon the initial startup of both circuits.

C.   CLOCK DIVISION

This section shows that precise, sequential data can be produced from X2 and output across the PC/104 bus as long as the circuit on X2 outputs data within the constraints of the maximum safe data rate. Producing sequential data from X2 is performed by implementing a circuit on X2 that runs at the same clock rate as the sampling rate on X1. In other words, ERR_RPT_TIME which sets the sampling rate should be equal to the clock division on X2, provided that X2 has already been clock divided to 25.5 MHz, for reasons discussed in section 2 below. As will be shown, using the reset signal discussed in Chapter III and equal clock-division on the two chips,

precise sequential data can be produced on the output across the PC/104 bus without any handshaking between the two chips.

1. Circuit Design

The circuit designed to demonstrate this synchronous capability of the two FPGA chips is the same circuit that is discussed in detail in Chapter V, An Example Experiment, and shown in the block diagram of Figure 10 below. This circuit, implemented on X2, employs the TMR design that has become commonplace within CFTP experiments. However, this circuit was also designed to provide an output that is easy to verify as correct or erroneous.
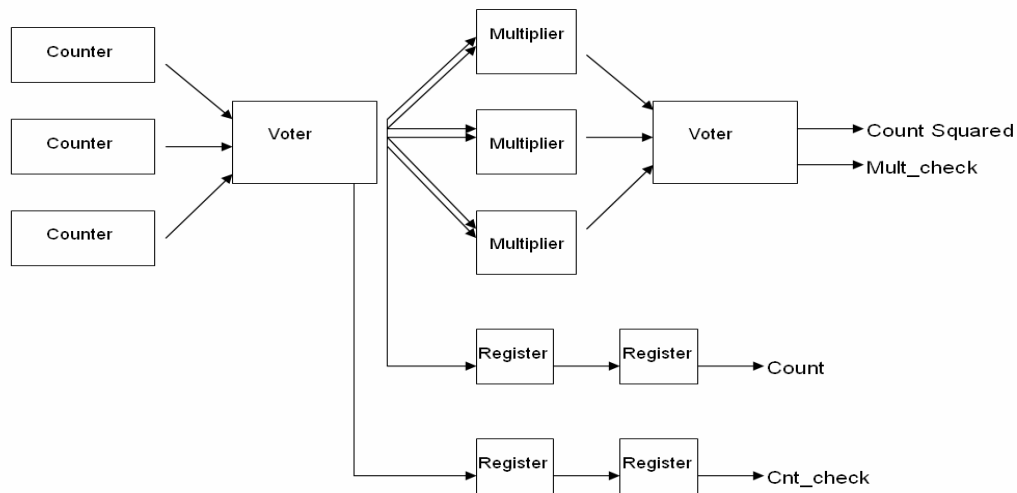
Figure 10.   TMR Multiplier

Though the circuit is discussed in Chapter V, a review of the output produced is necessary for this section. This circuit, named "TMR Multiplier," produces six distinct data outputs, four of which are illustrated in the block diagram of Figure 10, in the following order from left to right: error report that increments if either voter reports an error, a voter error report from the counter voter (Cnt_check), a voter error report from the multiplier voter (Mult_check), the counter output from the counter voter (Count), the multiplier output from the multiplier voter (Count Squared), and the standard timestamp produced by the top level module within X1.

The output of this circuit makes it easy to verify proper operation. The count can be squared, and that should equal the multiplier result.

### 2.	Clock Division

Because many of the modules on X1 run at 25.5 MHz, any circuit implemented on X2 must be initially clock-divided down to 25.5 MHz if the two chips are to be synchronized. For most experiments implemented on the CFTP architecture, X2 is not normally clock divided.   The following discussion on clock division within X2 is provided should future designers desire to run both chips at a reduced clock rate. This is required only if there is a desire to produce precise, sequential data from a circuit on X2, across X1, then across the PC/104 bus.

Synchronizing the two chips by speeding X1 back up to 51 MHz, as was done for the dual counters, would not allow the SelectMap processes to properly run (see Chapter III). Therefore, the initial clock signal coming into the circuit on X2 is clock-divided down to 25.5 MHz.   The timing diagram in Figure 11 provides a simple illustration of this clock division.
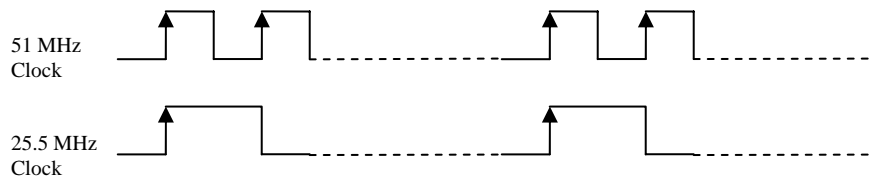
Figure 11.	25.5 MHz Timing Diagram.

The next level of clock division was set to match the value of the signal ERR_RPT_TIME located in x2Int.vhd. Clarification of how this signal works is required at this point. As mentioned in Chapter 3, clock-division on X2 is not required to get accurate data output across X1 and the PC/104 bus. What is required is that the constant signal ERR_RPT_TIME within x2Int.vhd is set to a value low enough to ensure the maximum safe data rate is not exceeded.   The constant signal ERR_RPT_TIME determines the sampling rate – the rate at which data is read within X1 and subsequently written to the PC/104 bus.

For this particular experiment, X2 is clock-divided twice; once to match the 25.5 MHz clock signal on X1, and again to match the sampling rate, set by adjusting ERR_RPT_TIME. As a result of these adjustments, functionally X1 is no longer sampling data; rather, X1 is reading and writing data at the rate of the clock on X2. The dual-counter experiment demonstrated that X1 and X2 could operate synchronously at 51 MHz. This experiment shows that the two chips can operate synchronously under equal clock-division.

3.   The Results

Several iterations of clock-division were implemented on the TMR Multiplier on X2, with each change accompanied by a change to the constant signal ERR_RPT_TIME in x2Int.vhd on X1. The results provide more solid evidence that the two chips can be synchronized, and shows that handshaking is not required to get precise, sequential data from a circuit on X2. Referring to Table 3, the results also provide more evidence that the maximum safe data rate for the CFTP architecture is approximately 500 Bps. The clock divisions shown in Table 3 are divisions on both chips beyond the initial division down to 25.5 MHz.

Clock Division	Effective Clock Rate	Bytes	Effective Data Rate (bytes/sec)	Observation
1,020,000	25 Hz	18	450	No errors, sequential count
10,200,000	2.5 Hz	18	45	No errors, sequential count
38,250,000	0.667 Hz	18	12	No errors, sequential count
76,500,000	0.337 Hz	18	6	No errors, sequential count
510,000	50 Hz	18	900	Missing data beyond hex 95

Table 3.   Data Rate Results from TMR Multiplier.

The output in Figure 12 was generated with both X1 and X2 operating at 25 Hz, which in this case equates to a data rate across the PC/104 bus of 450 Bps. Because the counters for the TMR Multiplier are designed to restart after reaching hex FF, it is

relatively simple to inspect the output data for errors. The data in Figure 12 was reviewed extensively, and no errors were noted.

Figure 12.   Output from TMR Multiplier at 450 Bytes/sec.

Figure 13 shows data produced by X2 at 900 Bps, significantly higher than the established maximum safe data rate. X1 was adjusted to read at the same rate, and the results show once again that the two chips can operate synchronously. The data in Figure 13 also shows that 900 Bps is in excess of the maximum safe data rate. Looking at the red circle, data is missing. This is the same symptoms noted from running the counter on X1 at a rate that is too high.

Figure 13.   Output from TMR Multiplier at 900 Bytes/sec.

4.   Sampling Data

In order to demonstrate that a circuit can be run on X2 at 51 MHz and its data collected by X1 running at 25.5 MHz, producing X2's data at an even slower sampling rate, the clock-division was removed from the TMR multiplier. Figure 14 below is the result of running the TMR multiplier at 51 MHz while its data was collected by X1 at only 0.667 Hz. To verify the correctness of this data, one of the results next to "Count" can be squared, and it will equal the result next to "Count Squared." Note that the results in Figure 14 are in hexadecimal format.

42

Figure 14.   TMR Multiplier at 51 MHz with Sampled Output at 0.667 Hz

5.   Final Analysis

The results from the experiments conducted in this chapter are significant. These results clearly show that the two FPGAs can be programmed to read and write synchronously without any handshaking. Specifically, X1 can be programmed to simply read data from specific pins at the same rate that X2 is writing data to those same pins, and X1 will record that data without error. These results therefore show that any clock skew present between the two chips is less than x ns, and that the clock skew inequality is not violated. Further, actual clock skew between the two chips is not large enough for future designers on the CFTP team to have to account for when creating experiments.

D.   CHAPTER SUMMARY

This chapter provided the required detail for designers to understand that timing with the CFTP architecture is a vital consideration when designing experiments. Specifically, the maximum safe data rate must be taken into account, and if precise, sequential data is desired from X2, the circuit on X2 must be clock-divided to match the

43

sampling rate on X1. The next chapter provides an example experiment, reviewing the process from beginning to end of how an experiment is implemented onto the CFTP architecture.

V. AN EXAMPLE EXPERIMENT

To aid future designers to fully understand the process by which an experiment is designed and implemented on the CFTP architecture, an example experiment is provided. This chapter will cover the basics, from beginning to end, of the processes by which an experiment is created, the Controller code modified, and output collected.

A. TRIPLE MODULAR REDUNDANCY

The overriding philosophy behind any experiment created within the CFTP team is reliability. Specifically, experiments are designed such that they not only can detect the occurrence of an SEU, but also they must be able to correct any erroneous data produced as a result of an SEU. Though specific implementations have varied, the primary method that designers have used to provide this reliability is TMR (triple modular redundancy). Circuits, or components within circuits, are produced in triplicate, and their outputs sent to a voter for comparison. As long as two of the three outputs agree, the data is considered reliable. This is how errors are detected and corrected. The voter identifies data that does not agree with two other streams of data, then decides what is reliable and identifies the component that provided unreliable data.

The circuit designed for this example experiment performs the functions described above; it employs TMR and provides data that shows if an error is detected and the specific component where that error occurred. However, this experiment was not designed specifically to test the applicability of TMR. This TMR Multiplier was designed with two goals in mind; one, to affectively demonstrate how a TMR experiment is implemented on the CFTP architecture, and two, to provide an output so that the CFTP team can verify proper operation of the Flight Board once in space.

B. TMR MULTIPLIER

One of the needs of the CFTP team is a circuit that is simple both in its operation and its output. It is important to be able to simply confirm that the two FPGAs on the Flight Board in space are operational. This circuit was designed with that requirement in mind. However, in the event that this becomes the only operational circuit in flight, TMR was included so reliability could still be evaluated. With these goals in mind, the TMR

Multiplier was designed to generate a count, square that count, and provide an output that verifies proper operation and the presence of any data errors.

Referring to Figure 15, this circuit, the TMR Multiplier, generates its own input via the use of a counter. The counter is produced in triplicate and the outputs voted. The voter produces two outputs; the count and a 3-bit data steam that identifies if any of the counters disagreed. It then takes this count and uses it for both inputs into a multiplier, thereby squaring the count. The multiplier was also produced in triplicate, those outputs fed to a voter, and two data streams are produced from that voter. As described in Chapter IV and shown in Figures 12, 13 and 14, five sets of data are produced by this circuit. This provides a data stream that simplifies the identification of errors, and provides easy confirmation of proper operation.

To further simplify the process of data verification, the counter not only counts in increments of one, base 16, from zero to "FF," but it then resets to zero and restarts the count. Section D provides details on the design of this circuit, to include how TMR works as well as the design decision behind the inclusion of registers, and Figure 15 should be referred to extensively when reading that section.

C. WORKING IN PROJECT NAVIGATOR

Though some of the important steps are included, the following is not intended as a manual for the use of Xilinx's Project Navigator [10]. A summary of the creation of the design within Project Navigator is included, but emphasis is on the specifics of the TMR Multiplier design followed by the required modifications of the Controller code to conform to the requirements of the experiment.
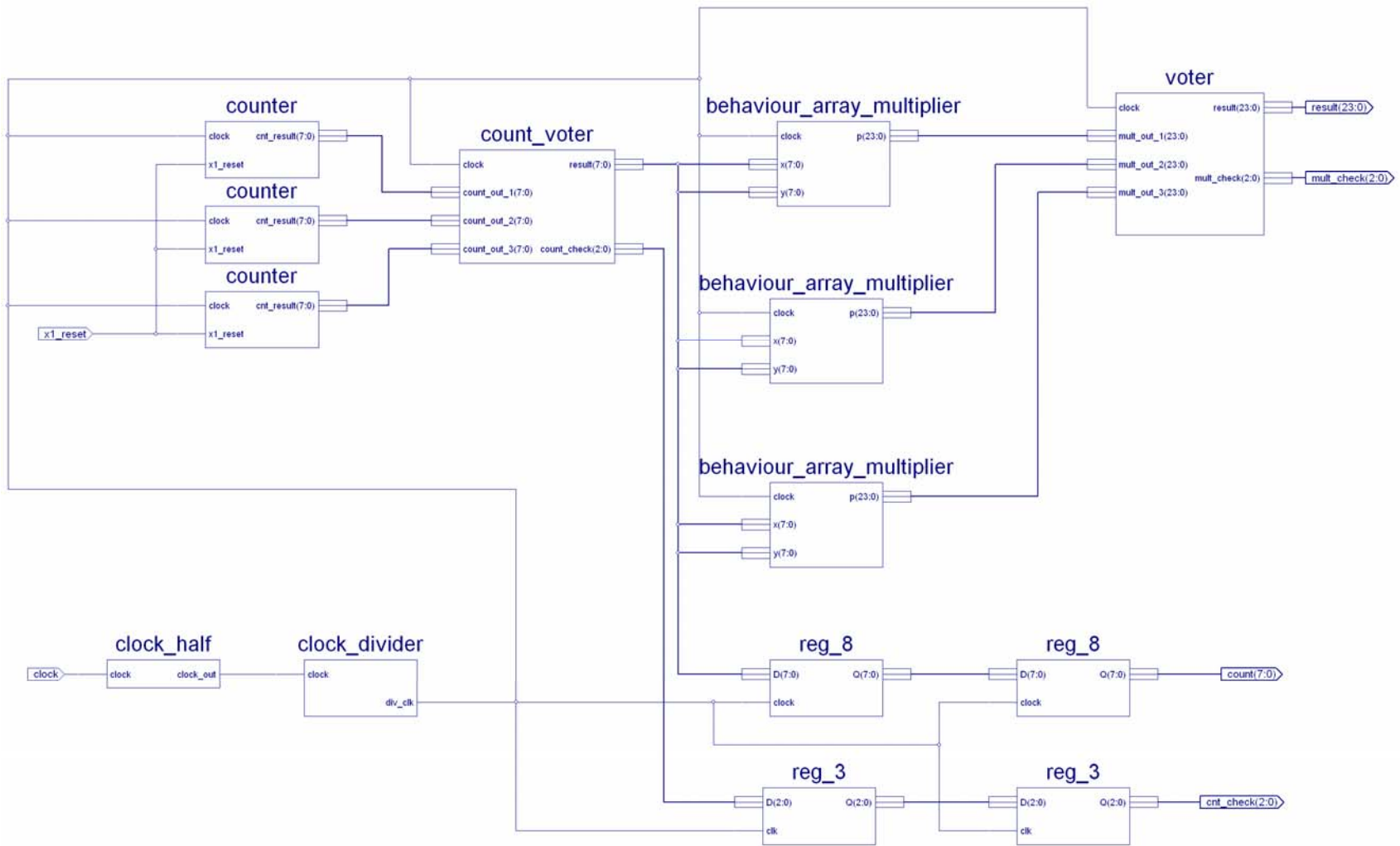
Figure 15.   TMR Multiplier Final Design

47

1. Creating a Design

To begin the design process for this TMR Multiplier, a project was created in Xilinx's Project Navigator. When creating a design, the user begins by going to "File," then selecting "New Project." After naming the project, it is important that the proper device and other options are chosen correctly. Looking at Figure 16, these are the options that should always be chosen for all projects created for the CFTP architecture, assuming the use of a Virtex I chip. Should future CFTP projects include the use of a later Virtex part (Virtex II or III or IV), then these options will change slightly.

For the current CFTP architecture, the options chosen are specific to the Virtex I chips; xqv600, cb228, and speed -4. These options are the same both the Flight and Development Boards. After this step is complete, the user will then be sent to a window and asked if a new source is to be created for the project. This step can be skipped and the user can add new sources in the Process View window.
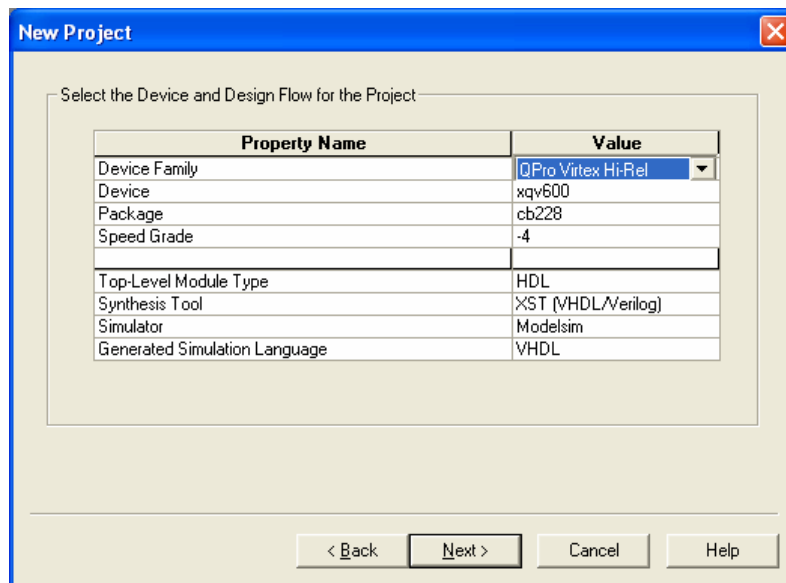
Figure 16.   Project Options in Xilinx's Project Navigator [10]

2. Processes in Project Navigator

In Figure 17, the process for editing/creating VHDL is shown, as well as the Process View window where sources can be added or created. In this Process View window are two processes of note for designers; Synthesize – XST, and Create Schematic Symbol. These two processes are also available for schematics. Once the designer has

48

finished editing a VHDL file, or wiring a schematic, the component then needs to be synthesized (compiled), and a schematic symbol created for addition to a higher level schematic. To perform either of these tasks, the specific component in the window, Sources in Project, must be highlighted.
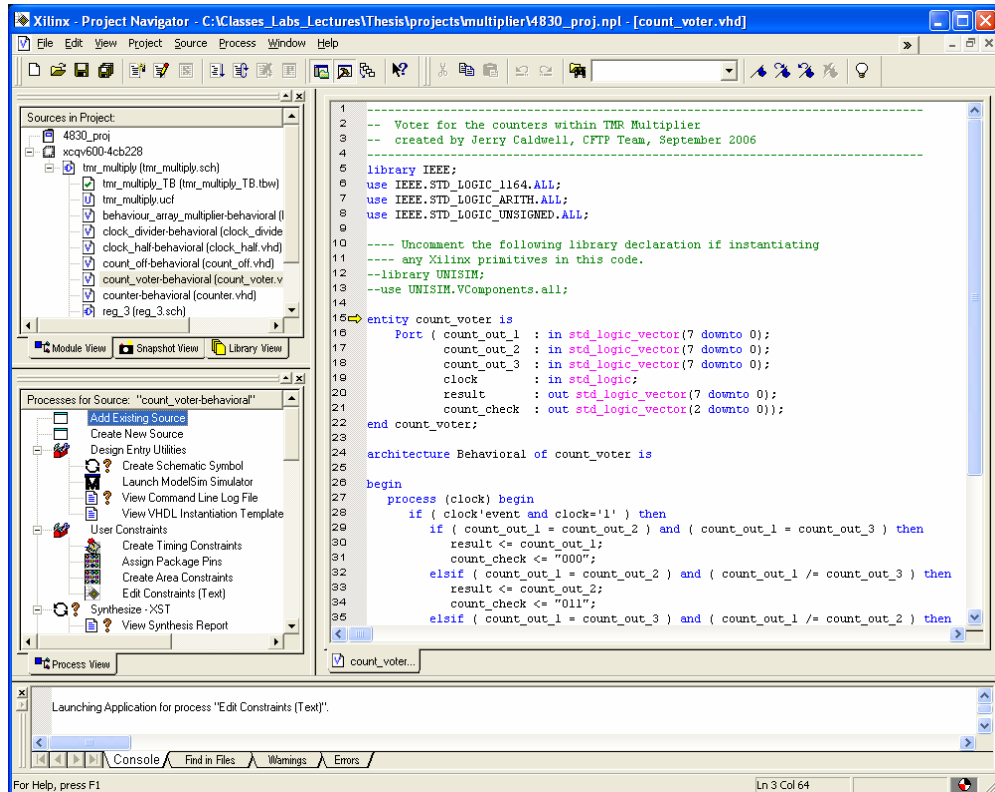
Figure 17.   Working with VHDL in Project Navigator [10]

The schematic for one of the registers from Figure 15 is included in Figure 18. Once all editing has been completed, and the schematic saved, the designer has to return to the window in Figure 16, highlight the component, and then synthesize and create a schematic symbol. The result is the symbol located in Figure 15, "reg_8." This symbol was added to the top level schematic by clicking on the tab labeled "Symbols," which can be seen in Figure 17.
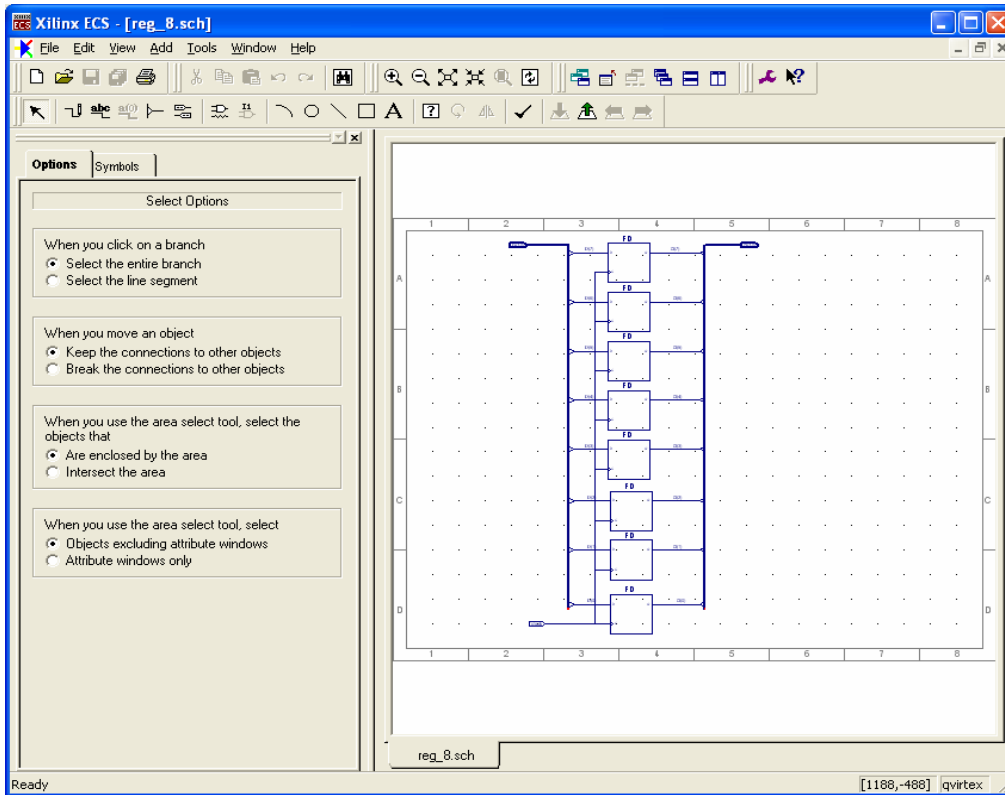
Figure 18.　　Schematic for Register in TMR Multiplier Design [10]

3.　　The Critical UCF Source

To ensure an experiment can properly interface with X1 in the CFTP architecture, the designer must include the constraint file discussed in Chapter III as a source to the project. This file is critical to the proper operation of an experiment within the CFTP architecture, because it identifies the FPGA pins with the signal names used in the design. The simplest way to get a constraint file is to copy one off of the CFTP server, then rename it specifically for the experiment (see Appendix A for detailed specifications of the path to the most current X1 ucf files for the Development and Flight Boards). For the TMR Multiplier design, the constraint file was re-named "tmr_mutliplier.ucf." Once this file was copied to the same directory where all the other files are located for the project, it was simply added as a source to the project.

Referring to Figure 19, once the constraint file has been added as a source to the project, it can then be edited by highlighting the file, then selecting "Edit Constraints (Text) in the process window. While editing this file, it is suggested that the designer

50

have a copy of the constraint file for X1 available to ensure that the pins within the respective files are properly matched with respect to functionality. See Appendix B for specifics.
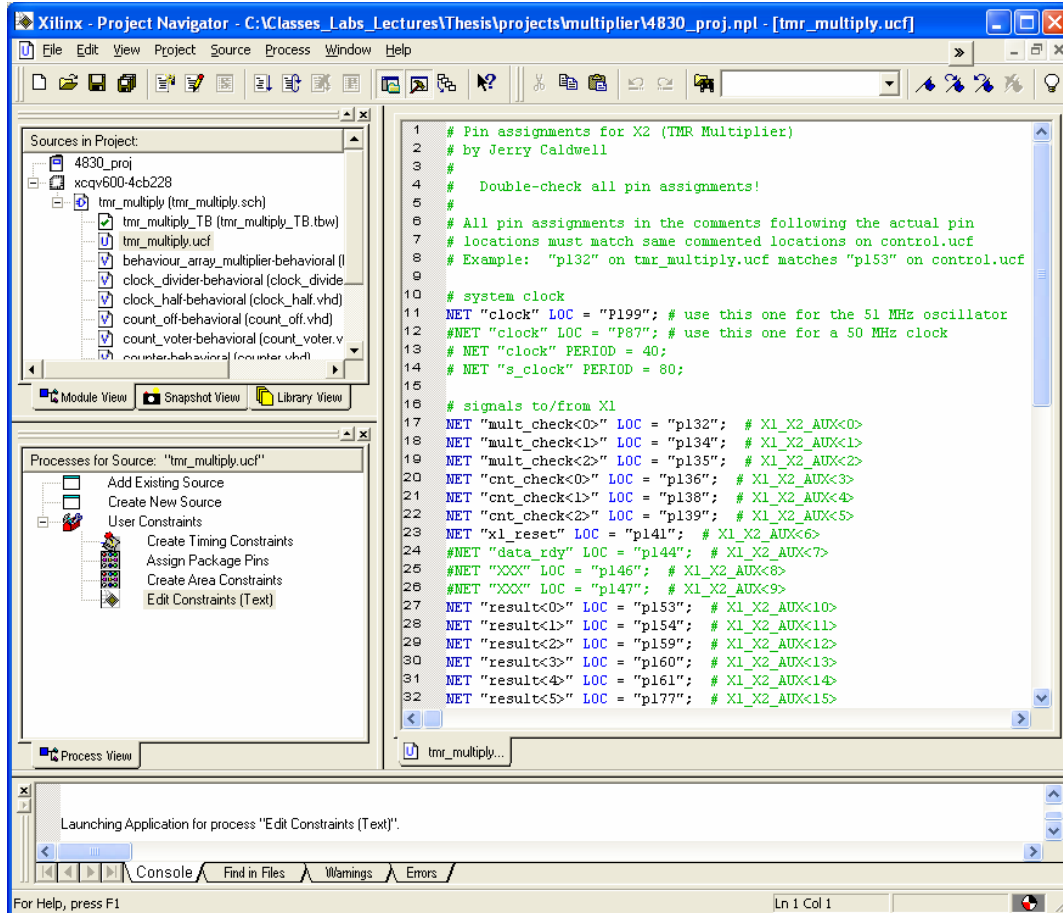
Figure 19.   A portion of the Constraint File [10]

D.   DETAILS OF THE EXPERIMENT

The design decisions behind the TMR Multiplier were driven by three important requirements, in addition to the goals stated in Section A: self-generated input, ability to synchronize the circuit with X1, and an output that is easy to verify as correct.

1.   Input & Synchronization

The counter provides self-generated input, and the TMR design is used on the counter to ensure a reliable input. Using a counter as the input also allowed for a simple way to synchronize the circuit with X1. As discussed in Chapter III, the top level code of X1 produces a reset signal upon the initial start up. Referring to Figure 20, this signal,

51

aptly named "x1_reset," is fed to each counter. The counter is designed to begin the count at zero when this reset signal goes high. Figure 21 shows the process for this, written in VHDL code.
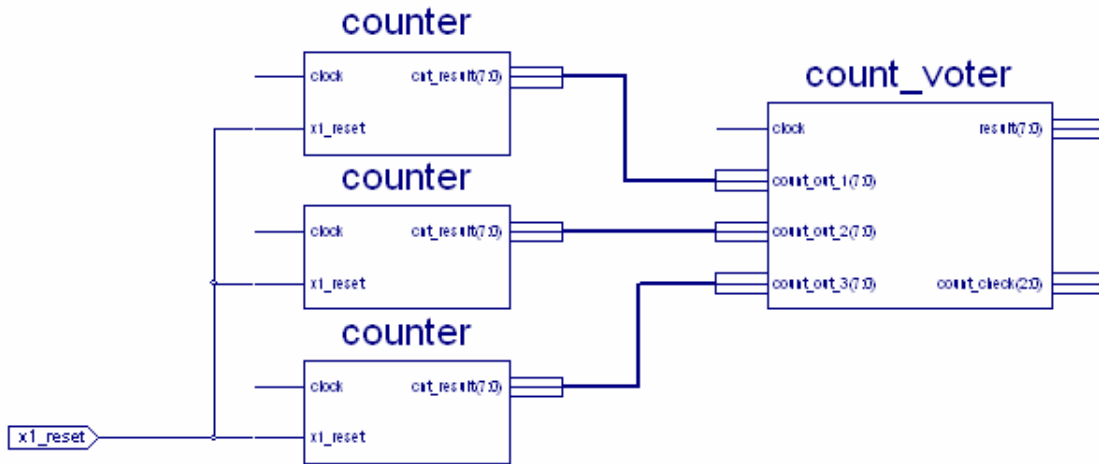
Figure 20.   Reset Signal and TMR Counter


```
20
21   architecture Behavioral of counter is
22
23   signal clock_cnt  : std_logic_vector(7 downto 0);
24
25   begin
26      process (clock, x1_reset) begin
27         if (x1_reset = '1') then
28            clock_cnt <= x"00";
29         elsif ( clock'event and clock='1' ) then
30            if ( clock_cnt = x"FF" ) then
31               clock_cnt <= x"00";
32            else
33               cnt_result <= clock_cnt;
34               clock_cnt <= clock_cnt + 1;
35            end if;
36         end if;
37      end process;
38   end Behavioral;
39
```

Figure 21.   VHDL Code for Counter.

2.   Voter Logic

The voters, both for the counter output and multiplier output, were also produced via VHDL code. Referring to the VHDL code in Figure 22, this is a simple process of

52

using "if statements" to determine which inputs agree, and assigning those to the output. This is data error correction, determining the output based upon an input where a majority (2 out of 3) agrees.

```
14
15⇨ entity count_voter is
16      Port ( count_out_1  : in std_logic_vector(7 downto 0);
17              count_out_2  : in std_logic_vector(7 downto 0);
18              count_out_3  : in std_logic_vector(7 downto 0);
19              clock        : in std_logic;
20              result       : out std_logic_vector(7 downto 0);
21              count_check  : out std_logic_vector(2 downto 0));
22   end count_voter;
23
24   architecture Behavioral of count_voter is
25
26   begin
27      process (clock) begin
28         if ( clock'event and clock='1' ) then
29            if ( count_out_1 = count_out_2 ) and ( count_out_1 = count_out_3 ) then
30               result <= count_out_1;
31               count_check <= "000";
32            elsif ( count_out_1 = count_out_2 ) and ( count_out_1 /= count_out_3 ) then
33               result <= count_out_2;
34               count_check <= "011";
35            elsif ( count_out_1 = count_out_3 ) and ( count_out_1 /= count_out_2 ) then
36               result <= count_out_1;
37               count_check <= "010";
38            elsif ( count_out_2 = count_out_3 ) and ( count_out_1 /= count_out_3 ) then
39               result <= count_out_3;
40               count_check <= "001";
41            else
42               result <= count_out_1;
43               count_check <= "111";
44            end if;
45         end if;
46      end process;
47   end Behavioral;
48
```

Figure 22.   Voter Logic

The portion of the voter that provides the location of the component which produced the error is located in the output vector "count_check."   If the voter logic detects that one of the inputs do not agree, or worse, that all three inputs disagree, then an appropriate signal is assigned to "count_check." For example, looking at the first "elsif" statement in Figure 22, if inputs one and two agree, but disagree with input three, then the output "result" is assigned with the majority input, and "count_check" is assigned the number three, signifying that counter number three is in error. This voter logic was also used to determine the multiplier output, with the only difference being the signal names.

53

3. Multiplier & Pipelining

The multiplier was created with one line of VHDL code; input "x" multiplied times input "y" is assigned to output "p."

$$p(23 \text{ downto } 0) <= x(11 \text{ downto } 0) * y(11 \text{ downto } 0);$$

This one simple line of code performs two functions. First, Xilinx's VHDL compiler will create a hardware multiplier, optimized by the compiler. Second, it automatically creates a register to hold the product until the next clock signal. This is important because, as per Chapter III, all outputs produced in X2 needs to be held in a register. Though the input from the counters are only 8 bits, they were converted to12 bits to provide a 24-bit output merely to aid in data stream formatting on the final output.

The VHDL compiler does not create a pipelined multiplier, and because the multiplier performs correctly at 51 MHz, pipelining internal to the multiplier was not required. However, pipelining was needed in the top level schematic. Referring back to Figure 14, two levels of registers were incorporated into the design. Because the output of the multipliers and the output of the second voter are held in registers, the output of the first voter had to be slowed down by two clock cycles to ensure all data produced from X2 arrived at X1 on the same clock.

To ensure complete reliability of the output of this circuit, the registers located in Figure 14 should also be produced in triplicate and the outputs voted. However, this was not done for two reasons; as mentioned, this experiment was not designed as validation of TMR, and two, even within the philosophy of TMR, at some point a decision must be made when TMR will not be incorporated to prevent overly complex circuits or circuits that are too large. Though not applicable to this circuit, at some point if too many components are duplicated, the design becomes too large and cumbersome. Additionally, registers take up only a small portion of the chip, so the probability of a data error due to an SEU within a register is minimal.

4. Signal Names

Referring to Figure 18, the tmr_multiply.ucf file, it is critical that the signal names in this constraint file exactly match final naming of signals in the top level schematic, Figure 14. This ensures that each signal is passed to the appropriate output pin on X2,

which enables the passing of proper data to X1 and ultimately across the PC/104 bus onto the ARM processor.

5. Sequential Data

In keeping with one of the initial stated goals, to verify proper operation of this circuit it is important that data is collected in the precise order is it produced. This provides verification that all components are functioning properly; the counters are verified if the count is produced sequentially, and the proper operation of the remaining components is verified via the normal TMR design principles.

To accomplish sequential output, as mentioned in Chapter 4, the circuit on X2 required the appropriate clock division. Two modules located in Figure 14 accomplish this. The module "clock_half" clock-divides the main clock down to the 25.5 MHz clock that drives the components on X1, and the module "clock_divider" can be adjusted to produce a signal at the same rate of the signal "ERR_RPT_TIME" located on X1, which is the sampling rate mentioned throughout this thesis.

6. Finishing the Experiment

The final step in Project Navigator, after the circuit has been simulated and checked for proper operation, is to "Implement" the design as discussed at the end of Chapter 3. This function performs four processes in the following order; translate, map, and then place and route (PAR). The translate process merges all of the input net-lists and design constraint information into one file. The map process maps the design to the FPGA, creating the ncd file, discussed in section 7, below. The PAR process takes this ncd file and performs the placing and routing of the design, connecting and routing all the wiring. The PAR process does not produce a different file extension; it modifies the existing ncd file. After this final step, the ncd file, in this case named multiply.ncd, will then be transferred to the CFTP server for two more processes, discussed in section 7, below.

7. Flash File

The last step for the experiment is to create a flash file, which is the configuration for X2 that will be stored on the flash memory module. This is performed on the CFTP server with two programs: bitgenpersist.sh, and mkflash.sh. The file that was transferred from the designer to the CFTP server has an "ncd" file extension. At the completion of

the last two processes, the experiment file will have a "fwr" file extension. The exact command line entries for these two processes are covered in detail in Appendix A.

The reason for executing bitgenpersist.sh and mkflash.sh is to create a bit file; a file that contains the configuration of the FPGA chip. The two processes accomplish this as bitgenperist.sh creates the bit file, and mkflash.sh strips off the headers, making the file ready for the loading of the configuration [1].

E.   MODIFYING THE CONTROLLER

As discussed in Chapter III, there are three VHDL modules within the Controller code (X1) that must be specifically modified by the designer; x2Int.vhd, top_level.vhd, and control.ucf. This section will list the specific code modified for the TMR multiplier with an accompanying explanation.

1.   X2 Interface

Within the file x2Int.vhd, the first changes to be made are the signal names going to or coming from X2, the experimental circuit. For the TMR Multiplier, only one signal goes to X2, the rest are signals coming into X1, the Controller. These signal names are located near the beginning of the code, and are easily identified as a result of the comments and the standard naming conventions discussed in Appendix A.

```
-- FOR EXPERIMENTAL DESIGN, signals coming from and going to X2
        DATA_TO_X2_RESET_o          : out std_logic;
        DATA_FROM_X2_OUTPUT_i    : in std_logic_vector(31 downto 0);
        DATA_FROM_X2_MULTCHK_i : in std_logic_vector(2 downto 0);
        DATA_FROM_X2_CNTCHK_i   : in std_logic_vector(2 downto 0);
-----------------------------------------------------------------------------------------------
```

The first signal name, "DATA_TO_X2_RESET_o," is the reset signal discussed in Chapter III that is generated via X1's top level code. This signal allows the circuits on X1 and X2 to begin at the same time.

The signal name, "DATA_FROM_X2_OUTPUT_i," is precisely what it implies; output generated from X2. Notice, however, that the end of the signal name has an underscore with the letter "i." In accordance with the naming conventions for CFTP, this signifies an incoming signal to X1. An underscore followed by the letter "o" was added to the outgoing reset signal as it is a signal that is outgoing from X1.

There are two specific streams of data within the 32 bits of the output signal name. This signal could have been broken up into those two streams of data, but for

56

programming simplicity, all the data was lumped into one 31-bit stream. Recall that the TMR Multiplier design on X2 outputs the count as well as the result of that count squared. The count is 8 bits and the square of the count is allotted 24 bits. As long as the designer knows what portion of the 32-bit data stream belongs to the count and to the multiplier, then it can be sorted out in the output data stored on the ARM processor.

This simplification of signal naming also allowed for simpler editing of the ucf file, which will be covered shortly.

Flowing down through the code in sequence, the next decisions to be made by the designer is the frequency for SelectMap read-backs and the data rate across the PC/104 bus.

```
CONSTANT   DLY_TIME        : integer := 765000000  -- 30 seconds
--CONSTANT   ERR_RPT_TIME  : integer := 76500000 -- (0.337 Hz)
CONSTANT   ERR_RPT_TIME  : integer := 38250000  -- (0.667 Hz)
--CONSTANT   ERR_RPT_TIME  : integer := 10200000  -- (2.5 Hz)
--CONSTANT   ERR_RPT_TIME  : integer := 1020000  -- (25 Hz)
```

The signal DLY_TIME is the rate at which a SelectMap read-back occurs. The standard within the CFTP development environment has been to leave this at 30-second intervals. It can be changed to suit the needs of specific experiments. The signal ERR_RPT_TIME is how the sampling rate is set (recall discussion from Chapter III). Above are four examples of data rates utilized for the output of the TMR Multiplier. Three of the sampling rates remain commented out of the code (in blue). The final one used for flight is uncommented and it generates an output every 1.5 seconds.

Recall also from Chapter III that the data rate across the PC/104 bus is determined by multiplying the sampling rate times the number of bytes to be transferred across the PC/104 bus per write cycle. The signal RPT_OUT_LENGTH is where this integer is set, and is located in the code just below where ERR_RPT_TIME resides.

```
CONSTANT   RPT_OUT_LENGTH  : integer := 18
```

Next the designer will modify the portion of the code in x2Int.vhd that actually writes data to the PC/104 bus. As illustrated with the code below, these signal names must exactly match the signal names near the beginning of the code in the port section. Notice there are 18, 8-bit words, assigned to an output vector. The first three words are usually not modified by designers. The "E," "R" and "00" are output merely for

formatting and identification purposes (E R identifies that relevant data follows). The fourth word is an error count, described shortly, and the next six words were added specifically for the TMR Multiplier. This portion of the data stream, highlighted in red, the number of words and specific signal names will vary to match specific experiments. The last eight words are a timestamp produced in the output stream, and is not modified by designers.

```
if ( report_out_vect = '0' and SM_CONFIG_STATUS_i = '0'
    and dly_timer = ERR_RPT_TIME ) then
    out_vect(0) <= x"45"; --E
    out_vect(1) <= x"52"; --R
    out_vect(2) <= x"00";
    out_vect(3) <= err_cnt(7 downto 0);  -- parameters set by designer
    out_vect(4) <= "00000" & DATA_FROM_X2_CNTCHK_i(2 downto 0);
    out_vect(5) <= "00000" & DATA_FROM_X2_MULTCHK_i(2 downto 0);
    out_vect(6) <= DATA_FROM_X2_OUTPUT_i(31 downto 24); -- counter output
    out_vect(7) <= DATA_FROM_X2_OUTPUT_i(23 downto 16); -- mult output
    out_vect(8) <= DATA_FROM_X2_OUTPUT_i(15 downto 8);  -- mult output
    out_vect(9) <= DATA_FROM_X2_OUTPUT_i(7 downto 0);   -- mult output
    out_vect(10) <= TIMESTAMP_i (63 downto 56); --timestamp
    out_vect(11) <= TIMESTAMP_i (55 downto 48); --timestamp
    out_vect(12) <= TIMESTAMP_i (47 downto 40); --timestamp
    out_vect(13) <= TIMESTAMP_i (39 downto 32); --timestamp
    out_vect(14) <= TIMESTAMP_i (31 downto 24); --timestamp
    out_vect(15) <= TIMESTAMP_i (23 downto 16); --timestamp
    out_vect(16) <= TIMESTAMP_i (15 downto 8); --timestamp
    out_vect(17) <= TIMESTAMP_i (7 downto 0); --timestamp
    report_out_vect <= '1';
---------Parameter for increasing the error count--------------------------
if ( (DATA_FROM_X2_MULTCHK_i /= "000") or (DATA_FROM_X2_CNTCHK_i /= "000")  ) then
        err_cnt <= err_cnt + 1;
    end if;
end if;
```

Signal names that correctly describe the output, like CNTCHK, do not require comments. However, to clarify what portion of the output is from the counter and the multiplier, comments were included next to the signals DATA_FROM_X2_OUTPUT_i. Specifically, the output of the counter occupies the top 8 bits of the 32-bit output stream, and the multiplier output occupies the remaining 24 bits.

The last portion of the code, located beneath the comment "parameters for increasing the error count," is important for two reasons; one, it provides data across the PC/104 bus that verifies a voter reported an error, giving the designer additional verification of a data error. Two, it counts the number of errors that occur so that a SelectMap reconfiguration can eventually take place when a set number of these data

errors have occurred. As can be seen above, the parameter for a data error from the TMR Multiplier is if one of the voters reports a number other than zero. Designers will have to modify the parameters within the IF statement specific to their experiments.

The threshold for the number of data errors before a reconfiguration is the last modification a designer needs to consider within x2Int.vhd. This threshold is located within an "if" statement, and is usually set to hex FF.

```
-- Set the threshold (# of data errors) for a reconfiguration
-- If we have 256 errors, reconfigure
         if ( err_cnt = x"FF" ) then
```

For most experiments, including the TMR Multiplier, this threshold can probably remain as is. However, designers are free to change this number if a specific experiment requires a higher or lower threshold before a SelectMap reconfiguration.

2.   The UCF File

A portion of the specific control.ucf file for the TMR multiplier design is included below, specifying the signal naming of data coming from and going to X2 that pertain to the TMR multiplier. The control.ucf file is significantly larger than the experiment.ucf file (tmr_multiplier.ucf) because of the various SelectMap and other pins. The complete control.ucf is located in Appendix B.

This specific control.ucf file is for the Development Board. The differences between the Development Board and Flight Board are not apparent here, but are covered in Appendix B.

```
NET "DATA_FROM_X2_MULTCHK_i<0>" LOC = "p153"; # X1_X2_AUX<0>
NET "DATA_FROM_X2_MULTCHK_i<1>" LOC = "p151"; # X1_X2_AUX<1>
NET "DATA_FROM_X2_MULTCHK_i<2>" LOC = "p150"; # X1_X2_AUX<2>
NET "DATA_FROM_X2_CNTCHK_i<0>" LOC = "p149"; # X1_X2_AUX<3>
NET "DATA_FROM_X2_CNTCHK_i<1>" LOC = "p147"; # X1_X2_AUX<4>
NET "DATA_FROM_X2_CNTCHK_i<2>" LOC = "p146"; # X1_X2_AUX<5>
NET "DATA_TO_X2_RESET_o" LOC = "p145"; # X1_X2_AUX<6>
#NET "DATA_FROM_X2_READY_i" LOC = "p144"; # X1_X2_AUX<7>
#NET "XXX" LOC = "p135"; # X1_X2_AUX<8>
#NET "XXX" LOC = "p134"; # X1_X2_AUX<9>
NET "DATA_FROM_X2_OUTPUT_i<0>" LOC = "p132"; # X1_X2_AUX<10>
NET "DATA_FROM_X2_OUTPUT_i<1>" LOC = "p127"; # X1_X2_AUX<11>
NET "DATA_FROM_X2_OUTPUT_i<2>" LOC = "p126"; # X1_X2_AUX<12>
NET "DATA_FROM_X2_OUTPUT_i<3>" LOC = "p120"; # X1_X2_AUX<13>
NET "DATA_FROM_X2_OUTPUT_i<4>" LOC = "p119"; # X1_X2_AUX<14>
NET "DATA_FROM_X2_OUTPUT_i<5>" LOC = "p112"; # X1_X2_AUX<15>
NET "DATA_FROM_X2_OUTPUT_i<6>" LOC = "p111"; # X1_X2_AUX<16>
NET "DATA_FROM_X2_OUTPUT_i<7>" LOC = "p110"; # X1_X2_AUX<17>
NET "DATA_FROM_X2_OUTPUT_i<8>" LOC = "p109"; # X1_X2_AUX<18>
```

```
NET "DATA_FROM_X2_OUTPUT_i<9>" LOC = "p108";  # X1_X2_AUX<19>
NET "DATA_FROM_X2_OUTPUT_i<10>" LOC = "p107";  # X1_X2_AUX<20>
NET "DATA_FROM_X2_OUTPUT_i<11>" LOC = "p105";  # X1_X2_AUX<21>
NET "DATA_FROM_X2_OUTPUT_i<12>" LOC = "p104";  # X1_X2_AUX<22>
NET "DATA_FROM_X2_OUTPUT_i<13>" LOC = "p103";  # X1_X2_AUX<23>
NET "DATA_FROM_X2_OUTPUT_i<14>" LOC = "p102";  # X1_X2_AUX<24>
NET "DATA_FROM_X2_OUTPUT_i<15>" LOC = "p101";  # X1_X2_AUX<25>
NET "DATA_FROM_X2_OUTPUT_i<16>" LOC = "p98";  # X1_X2_AUX<26>
NET "DATA_FROM_X2_OUTPUT_i<17>" LOC = "p97";  # X1_X2_AUX<27>
NET "DATA_FROM_X2_OUTPUT_i<18>" LOC = "p96";  # X1_X2_AUX<28>
NET "DATA_FROM_X2_OUTPUT_i<19>" LOC = "p94";  # X1_X2_AUX<29>
NET "DATA_FROM_X2_OUTPUT_i<20>" LOC = "p93";  # X1_X2_AUX<30>
NET "DATA_FROM_X2_OUTPUT_i<21>" LOC = "p92";  # X1_X2_AUX<31>
NET "DATA_FROM_X2_OUTPUT_i<22>" LOC = "p91";  # X1_X2_AUX<32>
NET "DATA_FROM_X2_OUTPUT_i<23>" LOC = "p90";  # X1_X2_AUX<33>
NET "DATA_FROM_X2_OUTPUT_i<24>" LOC = "p89";  # X1_X2_AUX<34>
NET "DATA_FROM_X2_OUTPUT_i<25>" LOC = "p88";  # X1_X2_AUX<35>
NET "DATA_FROM_X2_OUTPUT_i<26>" LOC = "p82";  # X1_X2_AUX<36>
NET "DATA_FROM_X2_OUTPUT_i<27>" LOC = "p81";  # X1_X2_AUX<37>
NET "DATA_FROM_X2_OUTPUT_i<28>" LOC = "p80";  # X1_X2_AUX<38>
NET "DATA_FROM_X2_OUTPUT_i<29>" LOC = "p79";  # X1_X2_AUX<39>
NET "DATA_FROM_X2_OUTPUT_i<30>" LOC = "p78";  # X1_X2_AUX<40>
NET "DATA_FROM_X2_OUTPUT_i<31>" LOC = "p77";  # X1_X2_AUX<41>
#NET "XXX" LOC = "p75"; # X1_X2_AUX<42> -- available on Flight Board
#NET "XXX" LOC = "p74"; # X1_X2_AUX<43> -- not avail on Flight Board
#NET "XXX" LOC = "p71"; # X1_X2_AUX<44> -- not avail on Flight Board
```

Notice that the signal names in the control.ucf file exactly match the signal names in the section "FOR EXPERIMENTAL DESIGN" in x2Int.vhd. Note also that in the constraint file, the pound symbol is used to comment out code, while in normal VHDL code, two dashes are used. This is an important note to show how unused pins are handled in the constraint file; simply comment them out of the code. Also, notice all the comments next to each pin declaration. As mentioned in Chapter III, these comments are located within both constraint files for X1 and X2. This is the method by which data is correctly declared and passed to the appropriate pins on both chips.

Once the ucf file has been properly edited, it is time to compile the X1 code.

3.   Makefile_Control

In the Linux environment, compiling of code, whether it be "C" code, or VHDL code, is performed via the command "make." Further, this process can be enhanced to suite the needs of specific projects by creating/editing a specific "makefile." This is the function of the Makefile_control and Makefile_experiment files for the CFTP team. The Makefile_control file is the last file the designer will need to modify.

```
# ID is used by the rd.sh program to determine how the output from
# your code should be formatted.  It is any 2 digit string
# Already taken:
# JS: Josh's Cordic
# JM: Jerry's Multiplier
# SR: James' Shift Register
# FD: Flash Dump
# VT: V2 Test code
# FE: Flash Erase
ID       = JM
DESCR       = "Jerry's Multiplier"
```

Above is the specific section of code from the file "Makefile_control" which must be modified by the designer. The two lines of code that have not been commented out need to be changed specific to the experiment, as per instructions in the commented section, with 2 digits (letters or numbers), followed by what they stand for. This is done for the purpose of "C" code that reads and formats output data. This "C" code is located in Appendix C.

4.   Compiling Code

At this stage, it is time to compile the X1 code and correct any noted errors by the compiler. Before beginning this process, all files and directories located within directory "control_out" must be deleted. This is performed with the following Linux command: rm -r *

Care must be exercised when using this command. It will delete ALL files and directories located within the directory where the command is issued. The contents of the control_out directory need to be removed as that is where the compiler sends history files from its previous compile. DO NOT use this delete command, "rm," in a directory above (higher level) control_out. Once this command is used the files CAN NOT be recovered. Ensure that this command is entered ONLY within the directory control_out.

After removing the contents of the directory control_out, the compile command is entered one level up from the control_out directory, where the Makefile_control file is located. The specific command is as follows: make -f Makefile_control

When the good fortune of a compile with no errors is achieved, then a file named "control.bin" exists in the control_out directory.

F. PROGRAMMING THE BOARD

Now it is time to physically program both chips and collect data. The designer has two important files, the fwr file for the flash, (X2's configuration), and the bin file which is X1's configuration. As noted previously, the fwr file was named to specifically identify the experiment, tmr_multiply_dev.fwr. This should be done as well for the control.bin file. For the multiplier, it was named, "tmr_mult_dev.bin."

Appendix A contains the specific procedures to program the chips and collect data for both the Development Board on the ground, and the Flight Board while on the Satellite.

G. CHAPTER SUMMARY

This chapter provided the overall procedures behind the development of an experiment for implementation onto the CFTP architecture. Most importantly, it covered the specific modifications that are required for the Controller code in order for experiments to properly interface with X1 and the PC/104 bus. The next chapter summarizes the work of this thesis and provides recommendations for future CFTP designers.

VI. CONCLUSIONS AND RECOMMENDATIONS

This thesis detailed the processes by which experiments are developed and implemented on the CFTP architecture. The structure of that architecture was discussed, as well as the inner-workings of the code that drives the Controller portion of the CFTP architecture. Finally, some of the limitations of the CFTP architecture were investigated, and an example experiment was detailed for the benefit of future CFTP designers.

A. SUMMARY

The CFTP architecture was designed around the framework of a key concept; two FPGA chips, one that implements fault tolerant experiments, and one that acts as a controller for the implementation of experiments and control of data produced from experiments. Significant components included to support this design are; a flash memory module, an EEPROM, a PC/104 bus, and an ARM Processor.

Within the CFTP environment, the Controller FPGA is named X1, and the FPGA for the implementation of experiments is named X2. Experiments implemented onto X2 can transfer data to and from X1 at the full rate of the CFTP oscillator, which is 51 MHz. However, the rate at which data can be transferred across the PC/104 bus is significantly less than 51 MHz due to the limitations of the ARM Processor and its ability to manage resources.

One of the most significant developments for the CFTP architecture was the VHDL code that creates the circuit on the Controller FPGA. This code is generic to the largest extent possible, which allows designers to make only minor changes such that X1 will interface properly with X2. The significance of the Controller code, beyond its ability to be modified specific to experiments, is its inherent ability to control the flow of experiments, compare configurations from what is stored in flash memory to what is running on X2, and perform a reconfiguration of X2 should a configuration error occur. It was the development of this code that made X1 a true controller and not just a pipe for data transfer, thanks to the tireless efforts of Mindy Surrat [1].

Despite the minor changes required to interface with an experiment, designers should become familiar with the Controller code. Specifically, a working knowledge of

all the VHDL modules, as well as the processes within the X2 Interface module, will give future CFTP designers a better understanding of the modifications required, and more importantly, will reduce the probability of making a change that creates an output of erroneous data.

Future designers for the CFTP team should also understand the limitations of the CFTP architecture, and how those limitations can potentially affect the design of an experiment. By knowing how the Controller interfaces with X2, designers have a greater chance of creating an experiment that produces results and provides insight to the viability of creating fault tolerant circuits for the space environment.

B.   CONCLUSIONS

The results discussed throughout this thesis were accomplished via detailed engineering analysis. The maximum safe data rate for the CFTP architecture was determined to be a result of the interactions between the ARM processor, the PC/104 bus, and X1 the Controller FPGA. Procedures to synchronize and clock-divide both FPGAs were investigated by implementing identical circuits on X1 and X2 and comparing their outputs. Finally, detailed mathematical and empirical analysis was employed to show that clock skew between the two FPGAs is manageable.

In addition to determining maximum safe data rate, the procedures for running experiments at 51 MHz and sampling output data were explored and documented. This provides future designers the necessary details to implement a myriad of designs such that they will properly interface with the components of the CFTP architecture.

C.   RECOMMENDATIONS

There is still work to be done within the CFTP architecture. Fortunately, some of the areas that still need to be explored can be done with software implementations, so having the Flight Board on a satellite in space is not a limiting factor.

1.   Use SDRAM Available to X2

As mentioned in Chapter II, 16 megabytes of RAM exist on the CFTP architecture. This RAM is available to X2, though it is has not yet been utilized in a formal experiment. Future designers should consider a use for this memory as this provides an expanded capability for potential fault tolerant designs.

2. Multiple Configurations on Flash Memory

The Flash Memory employed on the CFTP architecture has enough memory space to hold the configurations of four experiments. To date, the Controller only uses the first 900 KB of space on this memory module, loading one experiment on X2 and collecting data. The potential exists to modify the X1 code such that it can write four configurations to the Flash Memory, and then load an experiment onto X2, collect data for a set period of time, then load the next experiment. This would require modifications to the Flash Write Code, and as well as modifications to the primary Controller Code. This is an important capability of the CFTP architecture that should be explored as soon as possible.

3. Passing Data from the ARM

Currently, no process exists on the CFTP architecture that is capable of sending a data stream from the ARM processor to a circuit on X2 for processing. As a result, the only two methods the CFTP project has to provide input data to a circuit on X2 is to; one, create a circuit on X2 to generate the required data, as was done for this thesis (the TMR Multiplier), or two, implement a process on X1 to send the data to X2. This capability should be explored and implemented on the CFTP architecture. Circuits are generally designed to accept and process data, not to self generate data. Also, if possible, X1 should be left to perform its functions as a Controller and additional responsibilities added to X1 should be limited as much as possible.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: CFTP EXPERIMENT MANUAL

Appendix A is a manual designed as a "hands-on tool" for conducting CFTP experiments on the ground or in space. This manual assumes a level of knowledge by the user which, if not the case, references sections in Appendix B so the user can gain the necessary level of detail to understand the procedures contained herein.

This manual should also be viewed as "Standard Operating Procedures" for the CFTP team. This document contains standard naming conventions within the CFTP development environment for directories, file names, and signal names, which should be followed to the maximum extent possible. Many of the procedures listed were developed from years of lessons learned and therefore should be followed in detail.

Though Appendix B contains code for specific examples to aid in the sections throughout this manual, each Experimenter will also be given copies of previously operational code, as well as access to the CFTP server.

At the end of this manual is a checklist containing helpful reminders that can be used as an aid for running experiments on the ground, as well as a flow diagram. However, thorough familiarity of this manual is required in order to successfully run an experiment on either the development board or flight board.

A.   NAMING CONVENTIONS

Each experiment, and each modified version of the Controller Code, should reside in a set of directories with specific names. The top level directory for an experiment within the user's directory path should be the name of the respective project, for example, **tmr_multiplier**. This directory should contain two sub-directories, **dev_board** and **flight_board**. Both **dev_board** and **flight_board** should contain the files **Makefile_control**, **Makefile_experiment**, and sub-directories named **control_src**, **control_out**, **experiment_src**, and **experiment_out**. These names should not ever change. Only the specific file names within the experiment directories will change depending upon the name of the experiment.

A hierarchical representation of the directory structure is shown below:

```
…/proj_name/
        dev_board/
                Makefile_control
                Makefile_experiment
                control_src/
                control_out/
                experiment_src/
                experiment_out/
        flight_board/
                Makefile_control
                Makefile_experiment
                control_src/
                control_out/
                experiment_src/
                experiment_out/
```

Top-level directory ----------------- Project name

2nd -level directories ---------- dev_board   flight_board

files within 2nd –level directories   Makefile_control   Makefile_experiment

3rd -level directories --------------------------- control_src   control_out

3rd -level directories --------------------------- experiment_src   experiment_out
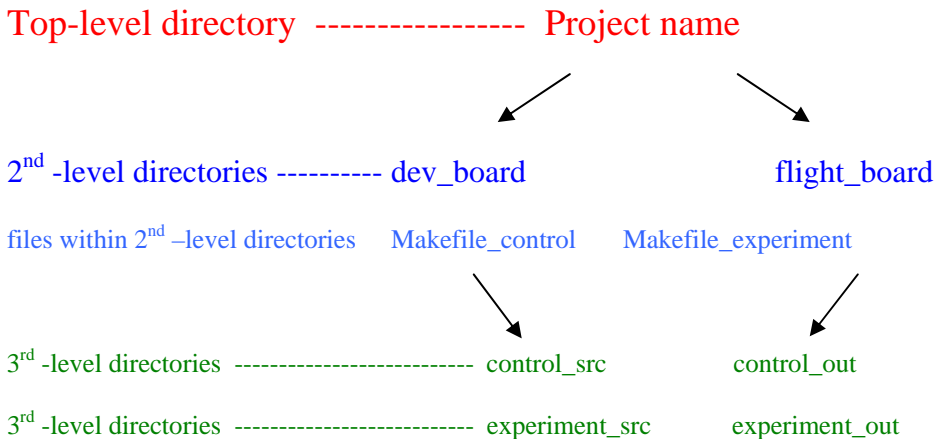
Linux directory tree examples – command-line prompt

$username/proj_name/dev_board/control_src

$username/proj_name/flight_board/experiment_src

B.   DEVELOPMENT BOARD & FLIGHT BOARD

Two different boards exist for the implementation and testing of experiments. One is named the "Development Board" and the other is the "Flight Board." The Flight Board is named as such because the FPGAs are identical to the ones in space, and therefore the pin layouts are identical. The Development Board has two FPGAs that are not designed for the space environment, and their differences, though minor, result in slightly different pin layouts between the two chips.

Appendix B addresses these differences in the constraint files. Throughout this manual, the Development Board is referred to as "dev_board" and the Flight Board as "flight_board" in accordance with the CFTP naming conventions.

C.   THE EXPERIMENT

1.   Simulation and Compilation

This phase of any experiment must be complete before beginning work within the CFTP development environment and modifying any X1 code. The standard program, for which the CFTP has a software license, is Xilinx's Project Navigator and ModelSim XE.

A working knowledge of Project Navigator and ModelSim, or another similar program, is up to the individual. If using Project Navigator, it is important that the following project properties within Project Navigator are set to the Xilinx part **xcqv600-4cb228**, whether designing for the Development or Flight Board.

a.   Naming Conventions

Specific experiments are where the greatest flexibility in file and signal naming exists. It is, however, important that file names for experiments represent what the experiment does, for example, **tmr_multiplier**. For the purpose of this manual, the word "experiment" will be used throughout.

b.   Constraint File

The constraint file, denoted with an "ucf" file extension, (experiment.ucf), is a critical portion of the experiment. If this file does not properly match the "ucf" file for the X1 controller code (control.ucf) then the experiment WILL NOT WORK.

Once the constraint file is properly written and added as a *source* in the respective project in Project Navigator and simulated with proper operation noted, the next step is to "implement" the design. This function in Project Navigator performs the necessary compiling, and then performs the required translate, map, place, and route portion for the code (schematics and/or VHDL) to run on an FPGA.

2.   Compiling within Linux ("make" files)

Note: Skip this part and go to part 3, "The NCD file," if using Xilinx's Project Navigator or other equivalent software. Refer to flow diagram at the end of the checklist.

The "make" files within the CFTP programming environment (e.g., **Makefile_experiment**) are designed to perform all the same functions that Project Navigator performs. This process of compiling, translating, mapping and routing should only be utilized on fully tested experiments which only require small modifications. It is highly recommended that Project Navigator, or an equivalent program, be used for the initial development of an experiment.

The advantage of compiling an experiment within the CFTP Linux environment is the consolidation of various procedures. The file "Makefile_experiment" performs "bitgenpersist.sh" and "mkflash.sh" immediately following the compiling and place and route operations. This allows the Experimenter to go directly to part 4, "Copy the fwr file," of this section. This method should only be used if enough experience has been gained such that a level of comfort exists within the CFTP development environment, as well as standard Linux operations.

a.   Modify the Makefile_experiment and experiment_prj files

The "experiment_prj" file lists the VHDL files and modules to be compiled. Simply modify a previously used "experiment_prj" file, ensuring that all required VHDL files for the respective experiment are listed in this file.

There is normally only one modification required within "Makefile_experiment." At the top, next to "ENTITYNAME =," needs to be the entity name of the top level code for the experiment. All VHDL code begins with "entity ……is," followed by the top level signals with the port declarations. The name following "entity" is what must be entered next to "ENTITYNAME" within the file "Makefile_experiment."

b. Compile

Compiling is performed one directory up from "experiment_src." All outputs generated from the compiling will go to the "experiment_out" directory. While in the dev_board or flight_board directory, the following command will be entered exactly as follows, with an example directory hierarchy followed by the "$" denoting a command prompt:

../project_name/dev_board$ make -f Makefile_experiment

If using this process (compiling within Linux), then once compiling is complete with no errors, skip to part 4, paragraph c, of this section, titled "Copy 'fwr' file for ground run."

3. The NCD file (experiment.ncd)

At the completion of the development and compilation phase (Part 1, Section A), an "ncd" file is created (**experiment.ncd**). The file now needs to be copied into the **experiment_out** directory of the CFTP programming environment.

4. Creating the Flash File

This is the last step in the experiment development phase. Perform the following steps in sequence.

a. Run bitgenpersist.sh

This command is entered exactly as follows with an example directory hierarchy followed by the "$" denoting a command prompt:

../project_name/dev_board$ bitgenpersist.sh experiment

As in all Linux commands, there must be at least one space between commands and parameters. Though you are performing this on a file with an ".ncd" extension, the extension is omitted when performing this operation. For example, if the

71

experiment name is "tmr_multiplier," and the file **tmr_multiplier.ncd** is the name of the file copied to the **experiment_out** directory, then the command would be as follows:

../tmr_multiplier/flight_board/experiment_out$ bitgenpersist.sh tmr_multiply

This command creates a number of files in the **experiment_out** directory, the same directory where the command was executed, but only two are needed. The two important files for the next part are the ".bin" and ".msk" files.

Note: "dev_board" and "flight_board" have been used, and will continue to be used, in examples to illustrate that the commands are the same for both directories.

### b.	Run mkflash.sh

This command creates the file that will be written to the flash, and is performed exactly as follows, with the "$" denoting a command prompt:

../experiment_out$ mkflash.sh experiment.bin experiment.msk > experiment.fwr

In this case the name of the output file must be entered with the ".fwr" extension. For example, if the experiment name is "tmr_multiplier," then the command would be as follows:

../flight_board$ mkflash.sh tmr_multiplier.bin tmr_multiplier.msk > tmr_multiplier.fwr

### c.	Copy "fwr" file for ground run

If running your experiment on the ground, then the ".fwr" file needs to be copied to the **/arm_mnt/flash_files/** directory on the CFTP server. It is important that the project name is unique as many ".fwr" files reside in this directory. If an experiment is to be compiled with different configuration files for the Development Board and the Flight Board, then consideration might be given to further appending the name during this copy process as such; "experiment_dev.fwr," or "experiment_flight.fwr."

## D.	THE CONTROLLER

### 1.	Compilation

The code for the controller (X1) does not need to be simulated, though doing so is not prohibited. This code has been developed and tested over time and is largely proven. Though the X1 code, specifically **x2Int.vhd** and **control.ucf**, are largely generic files, they must be modified to conform to the Experimental Design. At a minimum, the two

above files along with **top_level.vhd** will need to be modified to suit the needs of the Experiment. See Appendix B for specific details on the modifications of these programs.

Files within the **control_src** directory are listed below. The names of these files will never change:

- bitfile_V1.cmd
- clockGen.vhd
- control.ucf
- control.xcf
- control_prj
- impact.cmd
- pc104IntArm.vhd
- SelectMap_config.vhd
- SelectMap_readback.vhd
- top_level.vhd
- x2Int.vhd
- xstcmd.xst

a.   *Modify the Makefile_control file*

For the example shown in Appendix B, change "ID" to two initials that best reflect your experiment, and change "DESCR" to the exact name of your experiment. No other changes should be required.

b.   *Compile*

Compilation is performed one directory up from **control_src**. All outputs generated from the compile process will go to the **control_out** directory. While in the **dev_board** or **flight_board** directory, the following command should be entered exactly as follows, with an example directory hierarchy followed by the "$" denoting a command prompt:

../project_name/dev_board$ make -f Makefile_control

This process will create a ".bin" file in the **control_out** directory, named **control.bin**. This is the file that is used to program X1.

c.   *Copy the ".bin" file*

The file **control.bin** needs to be renamed to have the username appended on the end. An example of this Linux command is as follows:

../flight_board$ cp control.bin control_jerry_flight.bin

73

.   The renaming of this file is important as many different ".bin" files reside in the **/arm_mnt/arm_bin** directory. As noted when copying the **experiment.fwr** file, consideration should be given to modifying this filename according to its use for either the Flight or Development Board. The file should be copied to the **/arm_mnt/arm_bin** directory. An example "copy" command is as follows:

../flight_board$ cp control_jerry_dev.bin /arm_mnt/arm_bin

E.   GROUND RUN

1.   Naming conventions

Once   **proj_name.fwr**   and   **control_name_flight.bin,**   or (**control_name_dev.bin**), have been copied to the appropriate directory, there are a few more important naming conventions to discuss. The program **rd_arm_poll** will be used to read data output via the PC104 bus from the ARM processor. This output should be redirected to a file, and is done as follows:

./arm_bin/rd_arm_poll > filename

Because many files reside in the **/arm_mnt** directory, it is essential that this file be named as follows: **experiment_name_dev** or **experiment_name_flight**

Example: **multiplier_jerry_flight**

2.   ARM Commands via Telnet

At this point it is time to actually write the experiment to the flash and program X1 and X2. Before doing so, it is important to ensure that no one is using the ARM and programming X1 and/or X2.
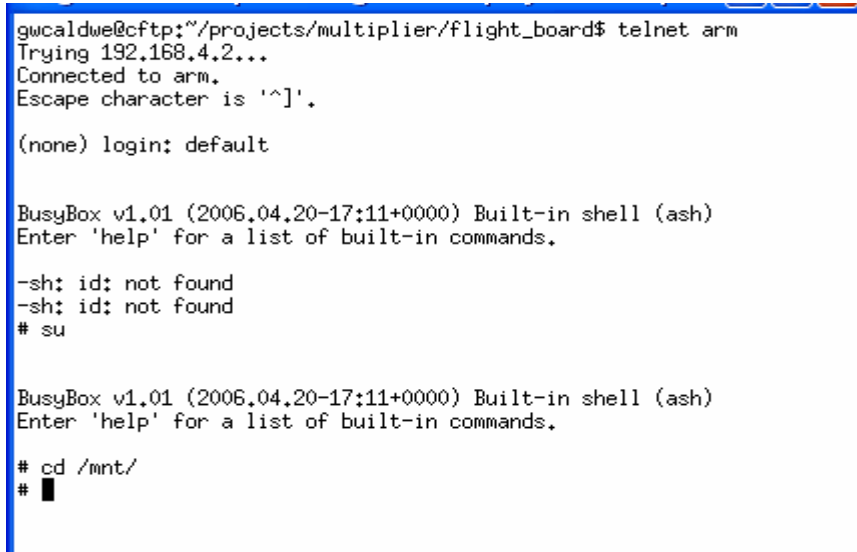
Note: While connected to the CFTP server, before beginning a "telnet" session to the ARM, the "who" command MUST be entered. Only one experiment can be programmed at a time. At a command-line prompt, a "w" can be entered and all users and their specific processes running will be listed. If any users are listed as connected to the ARM via "telnet," then exit and try again later.

Open three secure shell (ssh) windows. (The most commonly used ssh client within CFTP is "PuTTY.") Two of these windows will be dedicated to "telnet," and the other window is user preference, but normally the **/arm_mnt** directory is open in the

third window. The following commands must be entered exactly as below, from any directory within the CFTP server:

$ telnet arm
(none) login: default
 # su
 # cd /mnt/

The next figure is a screen shot depicting the typing of these commands to open a "telnet" window.


```
gwcaldwe@cftp:~/projects/multiplier/flight_board$ telnet arm
Trying 192.168.4.2...
Connected to arm.
Escape character is '^]'.

(none) login: default


BusyBox v1.01 (2006.04.20-17:11+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

-sh: id: not found
-sh: id: not found
# su

BusyBox v1.01 (2006.04.20-17:11+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

# cd /mnt/
#
```

a.   Running "write_flash.bin"

X1 must first be programmed to write an experiment to the flash. This is done with the following command, via the "telnet" window:

./arm_bin/jtag arm_bin/write_flash.bin

At the same time this command is run, in the other "telnet" window the **rd_arm_poll** should be run as follows:

./arm_bin/rd_arm_poll

When the **jtag** program is done programming X1 with **write_flash.bin**, a message will appear below the "rd_arm_poll" in "telnet" window 2.

```
X gwcaldwe@cftp: /home/gwcaldwe                    _ □ X
# ./arm_bin/jtag arm_bin/write_flash.bin

NOTICE: ACTIVE mode!

Setting up JTAG ... :
writing the bitstream...

word ffffffff ffffffff :11111111111111111111111111111111
word 665599aa aa995566 :10101010100110010101010101100110
word 01800030 30008001 :00110000000000001000000000000001
word 07000000 00000007 :00000000000000000000000000000111
word 01600130 30016001 :00110000000000010110000000000001
word 1d000000 0000001d :00000000000000000000000000011101
word 01200130 30012001 :00110000000000010010000000000001
word 2d3fa000 00a03f2d :00000000101000000011111100101101........
131072........
262144........
393216...
JSTART startup part... :
# █
```

```
X gwcaldwe@cftp: /home/gwcaldwe                    _ □ X
# ./arm_bin/rd_arm_poll

FLASH ERASE AND WRITE
LAST UPDATED:
WED JUL  5 16:47:20 PDT 2006
PART SUCCESSFULLY PROGRAMMED
█
```

Looking closely at the top of the two pictures, particular attention should be paid to exactly how the commands are entered. Notice the "./" before "arm_bin/jtag" and "arm_bin/rd_arm_poll."   In Linux this invokes the execution of the named program residing in the current directory.

b.   Running wr_arm_poll

Now that X1 is ready to write to the flash, it is time to do so. This command will program the flash with a file which was copied to the **/arm_mnt/flash_files** directory. In "telnet" window one, it is performed as follows:

./arm_bin/wr_arm_poll flash_files/experiment.fwr -i 10000

While this is running, rd_arm_poll should be run in "telnet" window 2 as done above during the **write_flash.bin** operation.

c.   Optional – running dump_flash.bin

This operation is performed to ensure that the flash was properly programmed. It is not required and should ONLY be used if problems exist with an experiment and verification of proper flash programming is desired.

In "telnet" window one, run the following command:

./arm_bin/jtag arm_bin/dump_flash.bin

In "telnet" window two, run the **rd_arm_poll** command with the output redirected to a uniquely named file. For example:

./arm_bin/rd_arm_poll > experiment_dump_flash

76
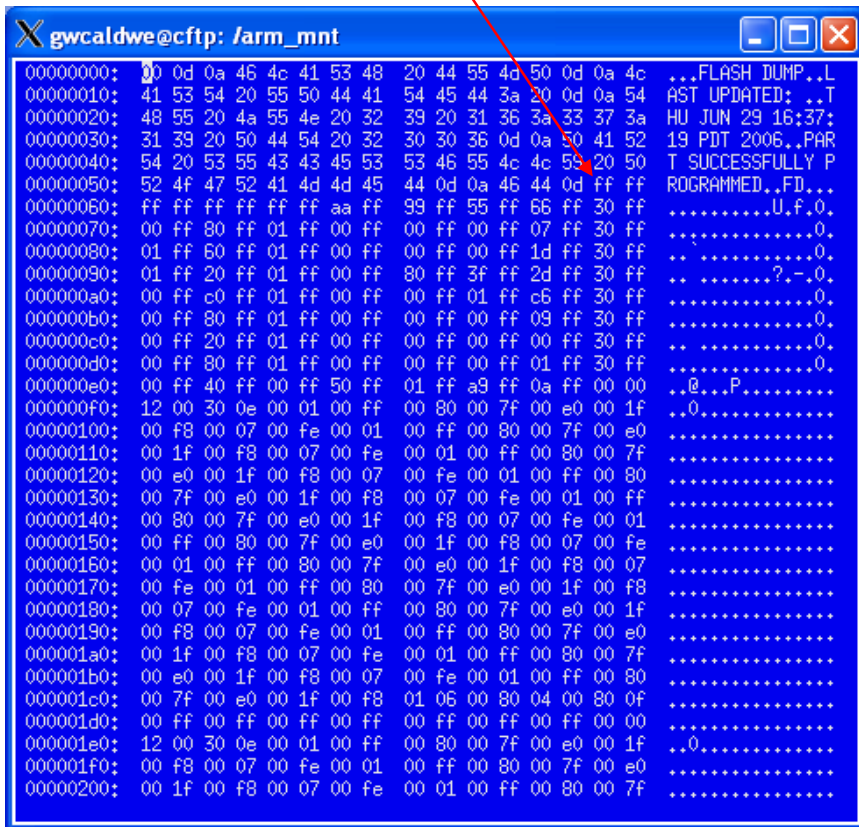
The **rd_arm_poll** process needs to continue to run once the dump flash portion is complete. However, in "telnet" window one, where **dump_flash.bin** just finished running, the file size of **experiment_dump_flash** needs to be monitored with the Linux command "ls –l filename." This should be done until the file size exceeds 900 KB, which occurs fairly quickly. See paragraph (d) of this section for an example of the "ls –l" command in Linux. Preventing the file size from growing much beyond 900 KB is not critically important. This is simply a good habit to maintain for reasons that will become apparent as experience is gained in this process.

Now compare the file **experiment_dump_flash** to **experiment.fwr** that was written to the flash. Before that can be done, the **experiment_dump_flash** file has to be slightly modified. This requires the use of a binary file editor. A good example of such an editor is the program **hexer**. It is a hex-editor program, which was used to produce the figure below. It is entered by typing the command

hexer filename
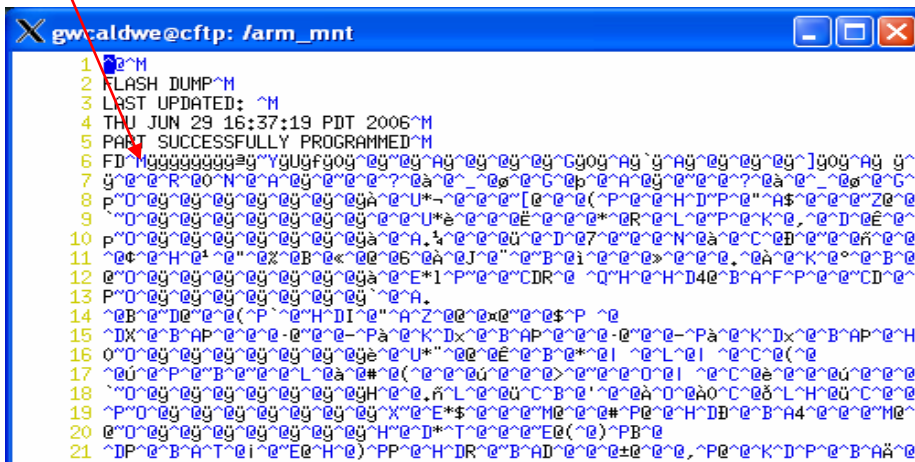
Open the experiment_dump_flash file and delete the first few lines all the way up to, but not including, the first "ff," as shown in the below figure with the red arrow. Use the "x" key to perform the delete operation. Saving the file is similar to saving a file in the Linux editor "vi." Enter a colon, which will give you a line at the bottom to enter another command. Enter a "w" followed by a space and a new filename. The file created by **dump_flash.bin** is read-only, therefore a new filename has to be created when saving the dump-flash file.

77

Delete all the way to here.

If performing this editing in "vi," delete all the way up to and including,
the "FD^M" symbols. Again, the file will have to be saved under a different name.

Now compare the two files. This is done with the executable **checkflash.sh** and is run as in the following example:

/arm_mnt$ checkflash.sh multiply_dump flash_files/tmr_multiply_dev.fwr

In the previous example, **/arm_mnt** is the current directory and the two filenames follow "checkflash.sh" with spaces in between. Running this will produce a long output. Continue pressing the space bar until the bottom of the output is reached. At address line 000dc360, there should be a string of 00's and ff's, as in the next figure with the red circle.

The **checkflash.sh** results, produced in the previous figure, serve as confirmation that the flash was properly programmed. If this precise line is not produced at the aforementioned line number, then that is evidence that the configuration was not properly written to the flash. Perform the wr_arm_poll operations in Section F, paragraph b, again, and then repeat this section on checking the flash configuration.

d.   *Running control_name.bin and collecting output*

Now it is time to program X1 once again, and finally collect your much anticipated output from your experiment. Listed below are both commands that need to be executed, one in each "telnet" window:

./arm_bin/jtag arm_bin/control_name_flight.bin

./arm_bin/rd_arm_poll > project_name_flight

After the **jtag** command completes its process, the experiment is running! If your programs operate properly, then data is collecting in the file **project_name_flight**. The rate at which the file increases in size needs to be monitored. This is done with the Linux command, "ls –l filename," which will give you the size of the file, along with other information.

Warning: Monitoring the file size is important during the first 30 seconds. If you have SelectMap readback enabled in your X1 code, then it is possible for this file to increase in size rapidly due to a SelectMap readback error.
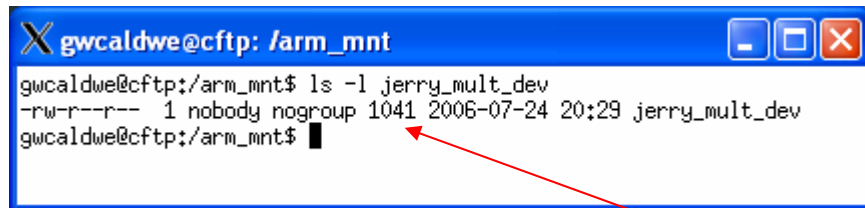
Size of file in bytes

The above screen shot is an example of what you will see after running the command "ls –l" on a specific file from the /arm_mnt directory. If the file size begins to rapidly increase 30 seconds after the **jtag** command completed its process, then the **rd_arm_poll** program needs to be immediately terminated by pressing Ctrl-C while in that window.

If after 30 seconds the file appears to be collecting data at an acceptable rate, then the output file can be viewed as it progresses with the command **hexdump**. This command should be executed from the window with the **/arm_mnt** directory open. It is entered as follows:

/arm_mnt$ hexdump -C project_name_flight | more

80

The "-C" option provides standard formatting to display 16 bytes per line, and the "more" command following the pipe character ("|") causes the output to be displayed a page at a time that fits the size of the screen. Press the space bar to scroll down to the next page of data.

Note: This hexdump command can be executed while **rd_arm_poll** is still running and outputting data to the same file that is being viewed via **hexdump**.

Data from a counter   Timestamp

F.   SATELLITE RUN

Once an experiment has been implemented on X2 with correctly operating code on X1, (on the Development Board), and data has been collected and verified, it is time to send the code for evaluation in space. Designers will work with the CFTP Research Associate to send the experiment.fwr and control.bin files to the satellite and implement their design on the Flight Board.

The procedures used to implement experiments on the Flight Board, while in orbit on the satellite, can also be used to implement an experiment on the Development Board. However, designers should first learn and use the "telnet" procedures contained within this Appendix to gain an appreciation of how the integration process occurs, and to aid in troubleshooting should errors in the implementation process surface.

Once an experiment has been successfully implemented via the "telnet" procedures and the designer has gained comfort with that process, then the procedures for implementing an experiment on the Flight Board should be practiced on the Development Board.

1. Implementing Experiments on the Satellite

To first practice implementing a design on the Development Board using the same procedures for the Flight Board in flight, the Development Board needs to be placed in the "flight mode." Contact the CFTP Research Associate to have the Development Board placed in flight mode.

Once the Development Board is in flight mode, programming the two chips is done via one command. This command is a program that takes all the telnet commands and streamlines them into one process. Before performing this command, the two files that program the two chips, the fwr file for X2 and bin file for X1, need to be moved to the same directory. The command, "load_flight_exper" is entered from the same directory where the two files are now located. The command is followed by the two files, experiment.fwr and control.bin, with a space in between, as seen in the example below:

$ load_flight_exper experiment.fwr control.bin

After a few minutes both FPGAs will be programmed and outputting data. The difference from the telnet procedures is that there will not be a uniquely named file collecting the output data. Data is now output into a generic file with a number appended on the end. The specific name of this output file will be provided by the CFTP Research Associate.

This program is how the Flight Board is programmed on the satellite. Once these procedures have been practiced on the Development Board, developers will work with the CFTP Research Associate to coordinate upload to the satellite.

G. CHECKLIST FOR RUNNING EXPERIMENTS

1. Run "bitgenpersist.sh" on the "ncd" file generated from Xilinx

../dev_board/experiment_out$ bitgenpersist.sh experiment

2. Run "mkflash.sh" on the "bin" and "msk" files created from "bitgenpersist.sh."

../experiment_out$ mkflash.sh experiment.bin experiment.msk > experiment.fwr

3. Copy the "experiment.fwr" file to the "/arm_mnt/flash_files" directory

4. Run "make –f Makefile_control" from one directory above "control_src."

5. Rename "control.bin" to "control_name_dev.bin" or "_flight," located in the "control_out" directory, and then copy said file to the "/arm_mnt/arm_bin" directory.

6. Perform the "who" command, or "w" to ensure the "arm" is not in use.

7. Open two additional "ssh" windows for a total of three.

8. Telnet to the "arm" in two of the windows.

9. Execute the "write_flash.bin" program in window 1, and the "rd_arm_poll" program in window 2.

Telnet Window 1: # ./arm_bin/jtag arm_bin/write_flash.bin

Telnet Window 2: # ./arm_bin/rd_arm_poll

10. Execute the "wr_arm_poll" program in window 1, and the "rd_arm_poll" program in window 2.

Telnet Window 1: # ./arm_bin/wr_arm_poll flash_files/experiment.fwr -i 10000

Telnet Window 2: # ./arm_bin/rd_arm_poll

11. OPTIONAL – run "dump_flash.bin" to check that the flash was properly programmed. This is more of a troubleshooting step than procedural.

12. Execute the "control_name_flight.bin" program in window 1, and the "rd_arm_poll > name_project" in window 2.

Telnet Window 1: # ./arm_bin/jtag arm_bin/control_name_flight.bin

Telnet Window 2: # ./arm_bin/rd_arm_poll > project_name_flight

83

13.   Monitor file size and output of file in the "/arm_mnt" window as "rd_arm_poll" runs in telnet window 2.

14.   Return to the "/arm_mnt" and run "hexdump" to view the output results.

Flow Diagram
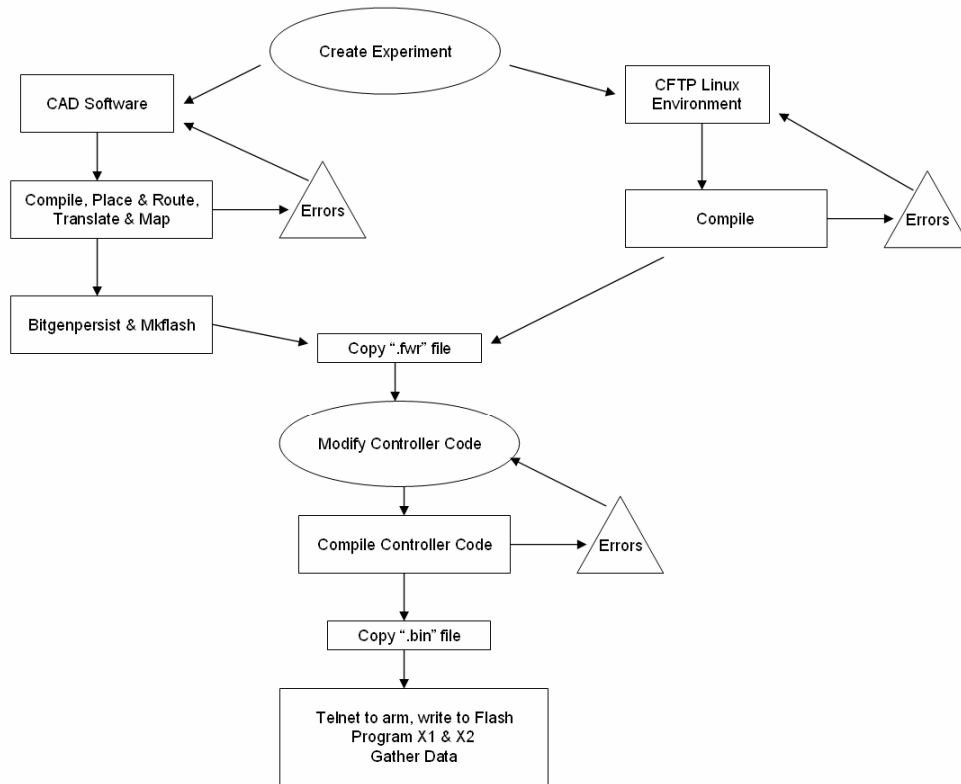
APPENDIX B: CONTROLLER CODE

This appendix contains important segments of code from the Controller. Specifically, the portions of the code that designers are required to modify or verify are included, and are highlighted in red. The entire code listing of x2Int.vhd is included, but only three specific portions of top_level.vhd because of its extensive length. All of the code from the control.ucf file for the development board is included, followed by the specific section for the flight board that differs.

TOP LEVEL

Only three areas within top_level.vhd need to be modified, and those coincide with signal naming. These signal names are located at the beginning of top_level.vhd in the port declaration section, and again in the component port declaration section for the x2Int module. The third area is specific signal assignment of the x2Int module located very near the bottom of top_level.vhd. All three areas are listed below – note the repeated naming of each signal for simplicity.

Also included is the approximate line number within the code where these areas are located. The comments are highlighted in blue to emphasize the location of these sections as the same blue comments appear in the code.

<u>Located near the top of the code, near line #52</u>

```
entity cftp_ARM is
  port (

      -- To/From X2 for Experimental Design, signals going to pins on X2
      -- change/add/remove as needed, also change control.ucf file to match
   DATA_TO_X2_RESET_o     : out std_logic;
   DATA_FROM_X2_COUNT_i   : in std_logic_vector (31 downto 0);
   DATA_FROM_X2_CNTCHK_i  : in std_logic_vector (2 downto 0);
```

<u>Located just below the above section, near line #122</u>

```
component x2Int port (
    CLOCK_i           : in std_logic; --50 MHz system clock
    RESET_i           : in std_logic;

    TIMESTAMP_i       : in std_logic_vector(63 downto 0);
```

85

```
    DATA_TO_X2_RESET_o     : out std_logic;
    DATA_FROM_X2_COUNT_i   : in std_logic_vector (31 downto 0);
    DATA_FROM_X2_CNTCHK_i  : in std_logic_vector (2 downto 0);
```

<u>Located very near the bottom of the code, near line #598</u>

```
x2Int0 : x2Int port map (

    CLOCK_i         => T_clock_i,       --: in std_logic;
    RESET_i         => ver_done_reset,      --: in std_logic;
    TIMESTAMP_i     => timestamp,
```

```
    DATA_TO_X2_RESET_o   => DATA_TO_X2_RESET_o,
    DATA_FROM_X2_COUNT_i => DATA_FROM_X2_COUNT_i,
    DATA_FROM_X2_CNTCHK_i => DATA_FROM_X2_CNTCHK_i,
```

X2 INTERFACE

The entire listing of x2Int.vhd is included below for the specific X1/X2 interface module used for the TMR multiplier. The portions highlighted in red are areas that require modification by designers for the CFTP team, and have been discussed throughout this thesis. The comments only for the areas that designers are required to modify are highlighted in blue to emphasize the location of these sections, and because the same blue comments appear in the code.

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

entity x2Int is
  port (
    CLOCK_i         : in std_logic;   -- 25.5 MHz signal
--     CLOCK_X2_i       : in std_logic; -- Additional clock if necessary
                        -- add appropriate clock to clockGen
    RESET_i         : in std_logic;   -- Reset signal
    TIMESTAMP_i     : in std_logic_vector(63 downto 0);
```

```
    DATA_TO_X2_RESET_o     : out std_logic;
    DATA_FROM_X2_OUTPUT_i   : in std_logic_vector(31 downto 0);
    DATA_FROM_X2_MULTCHK_i  : in std_logic_vector(2 downto 0);
```

DATA_FROM_X2_CNTCHK_i : in std_logic_vector(2 downto 0);

-- STANDARD TOP LEVEL SIGNALS - DO NOT CHANGE!
    STALL   : in std_logic;

    DATA_o   : out std_logic_vector(7 downto 0);-- Data bus out of X2 interface,
                              --in this case used to write to PC104
    DATA_i   : in std_logic_vector(7 downto 0); -- Data bus into X2 interface,
                              --in this case used to read from PC104

    PC104_WR_EN_o   : out std_logic; -- Active high, if WR_RDY = '1', then set WR_EN = '1'
                         -- for one clock and whatever is on DATA_o when WR_EN
                         -- is high will get written to PC104

    PC104_WR_RDY_i   : in std_logic; -- Active High, ok to write to PC104
                         -- if WR_RDY is high, whatever you write to PC104
                         -- will definitely get printed (your component has priority)

    PC104_RD_RDY_i   : in std_logic; -- Active high, if RD_RDY = '1',
                         -- then there is data on the PC104 bus ready to be read.
                         -- Once you read the data (from DATA_i), set RD_ACK high
                         -- for one clock to release the PC104

    PC104_RD_ACK_o   : out std_logic; -- Active high

    SM_CONFIG_RQST_o   : out std_logic; -- Active high, set config_rqst high for one clock if
                            -- you want to start a SelectMap config/reconfig

    SM_CONFIG_STATUS_i : in std_logic; -- Active high, stays '1' as long as a SelectMap config
                         -- is going on (don't request a readback or reconfig
                         -- while either is still active, it won't hurt anything,
                         -- but it won't go through)

    SM_RB_RQST_o   : out std_logic;-- Active high, set rb_rqst high for one clock if you
                         -- want to start a SelectMap readback

    SM_RB_STATUS_i   : in std_logic -- Active high, stays '1' as long as a SelectMap rb
                            -- is going on (don't request a readback or reconfig
                            -- while either is still active, it won't hurt anything,
                            -- but it won't go through)
    );
end x2Int;

architecture rtl of x2Int is

-- DLY_TIME counter and reset signals, might need adjustment to
-- meet the needs of an experiment
    CONSTANT DLY_TIME   : integer := 765000000; -- 30 seconds

    signal stall_d   : std_logic;
    signal s_reset_exp   : std_logic;
    signal first_reset   : std_logic;
    signal sm_rb_status_d   : std_logic;
    signal dly_cnt   : integer range 0 to DLY_TIME;
    signal dly_start_rb   : std_logic;

```vhdl
    signal sm_config_status_d   : std_logic;

-- error report timer signals
-- time between automatic error reports, adjust as needed
-- to meet specific needs of an experiment on X2
--    CONSTANT ERR_RPT_TIME      : integer := 76500000; -- (0.33337 Hz = 3.0 sec)
      CONSTANT ERR_RPT_TIME      : integer := 38250000; -- (0.66667 Hz = 1.5 sec)
--    CONSTANT ERR_RPT_TIME      : integer := 10200000; -- (2.5 Hz, 45 Bps)
--    CONSTANT ERR_RPT_TIME      : integer := 1020000; -- (25 Hz, 450 Bps)
-- set to the number of bytes included in your output vector that needs
-- to be printed to PC104
      CONSTANT REPORT_OUT_LENGTH  : integer := 18;
    signal dly_timer          : integer range 0 to ERR_RPT_TIME;
    signal count_out_vect       : integer range 0 to REPORT_OUT_LENGTH;
    signal report_out_vect      : std_logic;

    type output_vector
       is array(REPORT_OUT_LENGTH-1 downto 0)
       of std_logic_vector(7 downto 0);

    signal out_vect            : output_vector;

--readback/reconfig process
    -- For external JTAG error injection, we must pause for a time before
    -- trying to readback/reconfig.  The part can become active before
    -- programming is complete, and errors can start accumulating.  Readback
    -- DOES NOT work while JTAG is active.  This can be 0 when we're not using
    -- JTAG error injection.
    CONSTANT DLY_RECONFIG          : integer := 153000000; -- 6 seconds
-- readback/reconfig process, CLOCK_X2_i signals
    signal err_cnt            : std_logic_vector(23 downto 0);
    signal exp_start_rb          : std_logic;
    signal reconfig_from_error       : std_logic;
    signal reconfig_from_error_save    : std_logic;
    signal rb_started            : std_logic;
    signal reconfig_timer           : integer range 0 to DLY_RECONFIG;


begin
-- Asynchronous assignments of top level signals
    DATA_TO_X2_RESET_o <= s_reset_exp;
    SM_RB_RQST_o <= exp_start_rb or  dly_start_rb;
    SM_CONFIG_RQST_o <= reconfig_from_error;

-- Timer to determine how frequently to print out heart beat error reports
    process(CLOCK_i, s_reset_exp) begin
      if (s_reset_exp = '1') then
        dly_timer <= 0;
      elsif(CLOCK_i'event and CLOCK_i = '1') then
        if (dly_timer = ERR_RPT_TIME) then
          dly_timer <= 0;
        else
          dly_timer <= dly_timer + 1;
        end if;
      end if;
    end process;
```

```vhdl
-- Do reset for experiment
   process (CLOCK_i,RESET_i) begin
      if (RESET_i = '1') then
         s_reset_exp <= '1';
         stall_d <= '0';
      elsif (CLOCK_i'event and CLOCK_i = '1') then
         sm_config_status_d <= SM_CONFIG_STATUS_i;
         stall_d <= STALL;
         s_reset_exp <= '0';
--              if (first_reset='0' or (stall = '0' and stall_d = '1') or (SM_CONFIG_STATUS_i = '0' and
sm_config_status_d = '1' ) ) then
-- MLS 2005.11.27 taking out reset after stall!  hopefully will have partial active
-- reconfig working so it won't be necessary
         if (first_reset='0' or (SM_CONFIG_STATUS_i = '0' and sm_config_status_d = '1' ) ) then
            s_reset_exp <= '1';
         end if;
      end if;
   end process;

--MLS 2005.09.15 wait until after X2 is done configuring to do first reset (instead of 30s)
   process(CLOCK_i,RESET_i) begin
      if (RESET_i = '1') then
         first_reset <= '0';
      elsif (CLOCK_i'event and CLOCK_i = '1') then
         if (first_reset = '0' and SM_CONFIG_STATUS_i = '0' and sm_config_status_d = '1') then
            first_reset <= '1';
         end if;
      end if;
   end process;

-- Every DLY_TIME clocks after that, do a SelectMap readback
--don't start dly_cnt until s_reset_exp goes low (after version is done and x2 config is done)
   process(CLOCK_i,s_reset_exp) begin
      if (s_reset_exp = '1') then
         sm_rb_status_d <= '0';
         dly_cnt <= 0;
         dly_start_rb <= '0'; --Signal to notify that the delay counter wants to perform a SMRB
      elsif(CLOCK_i'event and CLOCK_i = '1') then
         dly_start_rb <= '0';
         sm_rb_status_d <= SM_RB_STATUS_i;

-- If we just finished a SMRB, start the counter over
         if (SM_RB_STATUS_i = '0' and sm_rb_status_d = '1') then
                 dly_cnt <= 0;
         elsif (dly_cnt < DLY_TIME) then
            dly_cnt <= dly_cnt + 1;
         elsif (dly_cnt = DLY_TIME) then
            dly_cnt <= 0;
            dly_start_rb <= '1';--After DLY_TIME clocks, start a readback
         end if;
      end if;
   end process;

--Process to write error reports out to the PC104
   process (CLOCK_i, s_reset_exp) begin
```

89

```vhdl
--hold off printing error reports until experiment is reset
     if (s_reset_exp = '1') then
        count_out_vect <= 0;
        report_out_vect <= '0';
        PC104_WR_EN_o <= '0';
        DATA_o <= x"31";
        err_cnt <= x"000000";
     elsif (CLOCK_i'event and CLOCK_i = '1') then
        PC104_WR_EN_o <= '0'; --default assignment for WR_EN
--          divided_clock <= DATA_FROM_X2_CLKDIV_i;


-- Whenever we've gone through ERR_RPT_TIME clocks or we get an error from X2
-- (a signal coming directly from X2), and we've already finished printing out
-- the last error report (report_out_vect = '0'), set the output error report
-- vector to the correct values  MLS debug change back to t_err_from_x2_i = '1'
        if ( report_out_vect = '0' and SM_CONFIG_STATUS_i = '0'
           and dly_timer = ERR_RPT_TIME ) then
           out_vect(0) <= x"45"; --E
           out_vect(1) <= x"52"; --R,
           out_vect(2) <= x"00";
           out_vect(3) <= err_cnt(7 downto 0);
           out_vect(4) <= "00000" & DATA_FROM_X2_CNTCHK_i(2 downto 0);
           out_vect(5) <= "00000" & DATA_FROM_X2_MULTCHK_i(2 downto 0);
           out_vect(6) <= DATA_FROM_X2_OUTPUT_i(31 downto 24); -- counter output
           out_vect(7) <= DATA_FROM_X2_OUTPUT_i(23 downto 16); -- mult output
           out_vect(8) <= DATA_FROM_X2_OUTPUT_i(15 downto 8);  -- mult output
           out_vect(9) <= DATA_FROM_X2_OUTPUT_i(7 downto 0);   -- mult output
           out_vect(10) <= TIMESTAMP_i (63 downto 56); --timestamp
           out_vect(11) <= TIMESTAMP_i (55 downto 48); --timestamp
           out_vect(12) <= TIMESTAMP_i (47 downto 40); --timestamp
           out_vect(13) <= TIMESTAMP_i (39 downto 32); --timestamp
           out_vect(14) <= TIMESTAMP_i (31 downto 24); --timestamp
           out_vect(15) <= TIMESTAMP_i (23 downto 16); --timestamp
           out_vect(16) <= TIMESTAMP_i (15 downto 8); --timestamp
           out_vect(17) <= TIMESTAMP_i (7 downto 0); --timestamp

           report_out_vect <= '1';
           if ( (DATA_FROM_X2_MULTCHK_i /= "000") or (DATA_FROM_X2_CNTCHK_i /= "000")
) then
                err_cnt <= err_cnt + 1;
           end if;
        end if;
-- If we've set the output vector (report_out_vect = '1'), then print the output vector to the PC104
-- one byte at a time (REPORT_OUT_LENGTH bytes will be printed)
-- Be sure to set REPORT_OUT_LENGTH to proper value in signal definitions above
        if (report_out_vect='1') then
           if (count_out_vect < REPORT_OUT_LENGTH and PC104_WR_RDY_i = '1') then
              DATA_o <= out_vect(count_out_vect);
              PC104_WR_EN_o <= '1'; --write byte
              count_out_vect <= count_out_vect + 1;
           elsif (count_out_vect = REPORT_OUT_LENGTH) then
              count_out_vect <= 0;
              report_out_vect <= '0';
           end if;
        end if;
     end if;
```

```vhdl
   end process;

-- Process to signal SM RB/RC from an experiment error.
-- Note that if you have your experiment running at a different speed than
-- 25 MHz (the speed of CLOCK_i), you must be VERY careful about moving
-- between clock domains.  Basically have another process on your experiments'
-- clock that sets a flag to trigger a readback, then put that signal on the
-- 25 MHz clock in this process (see Josh's x2Int.vhd for an example)
   process(CLOCK_i, s_reset_exp) begin
      if (s_reset_exp = '1') then
         reconfig_from_error <= '0';
         exp_start_rb <= '0';
         rb_started <= '0'; --make sure exp_start_rb only 1 clock
      elsif (CLOCK_i'event and CLOCK_i = '1') then
         exp_start_rb <= '0';
         reconfig_timer <= reconfig_timer + 1;
         reconfig_from_error <= '0';

-- Set the threshold (# of data errors) for a reconfiguration
-- If we have 256 errors, reconfigure
         if ( err_cnt = x"FF" ) then
            reconfig_from_error_save <= '1';
            reconfig_timer <= 0;

-- Wait until SMRB is done, and then request a reconfig from the top level
         elsif ( reconfig_from_error_save = '1'
             and SM_RB_STATUS_i = '0'
             and sm_rb_status_d = '1' ) then
            reconfig_from_error <= '1';
            reconfig_from_error_save <= '0';
            rb_started <= '0';

-- Wait until last error report is printed out before starting readback
-- Once readback has started, don't start another one!
-- Wait to start readback for ~3s (ERR_RPT_TIME) after reaching critical
-- number of errors, this is for JTAG external error injection, errors
-- begin accumulating before it is done programming, so 128 errors could
-- be reached before partial reconfig complete, so tries to readback
-- while JTAG still going on.
         elsif (reconfig_from_error_save = '1'
             and reconfig_timer = DLY_RECONFIG
             and report_out_vect = '0'
             and SM_RB_STATUS_i = '0'
             and rb_started = '0' ) then
            exp_start_rb <= '1';
            rb_started <= '1';
         end if;
      end if;
   end process;
end rtl;
```

CONTROL.UCF – DEVELOPMENT BOARD

The entire code listing for X1's constraint file used for the TMR multiplier is included below. Immediately following is X2's constraint file for the TMR multiplier. Notice the blue comments on X2's constraint file for comparison with X1's constraint file and how those comments denote how the pins correspond between the two chips.

```
# Pin assignments for X1 - development board
# Jerry Caldwell's TMR Multiplier
# Created 21 August 06.
#
# Modified on 22 July 06 for use on the Development Board.
#
# Aux pins 42 to 44 were added, which are not available on the
#   flight board, many other pin changes are different from the
#   flight board, and are notated in comments to the right of the
#   affected pin assignments
#
# All pin assignments in the comments following the actual pin
#   locations must match same commented locations on cftp_x1.ucf
#   Example:  "p153" on control.ucf matches "p132" on experiment.ucf
#
# The following are new Pin Assignments for the Flight Configuration
#   where the PC104 bus is used with a JTAG interface - these are not
#   use very often.

#NET "T_CARD_BLEO_i" LOC = "p44";  # only needed for writing low byte
#NET "CARD_BLE1" LOC = "p12";
#NET "CARD_RESET" LOC = "p54";
#NET "CARD_DATA_HIGH<10>" LOC = "p24";
#NET "CARD_DATA_HIGH<11>" LOC = "p26";
#NET "CARD_DATA_HIGH<12>" LOC = "p31";
#NET "CARD_DATA_HIGH<13>" LOC = "p33";
#NET "CARD_DATA_HIGH<14>" LOC = "p35";
#NET "CARD_DATA_HIGH<15>" LOC = "p38";
#NET "CARD_DATA_HIGH<8>" LOC = "p20";
#NET "CARD_DATA_HIGH<9>" LOC = "p22";
#NET "CLOCK_OUT" LOC = "p70";

NET "T_VPPEN_o" LOC = "P60";       # only needed for writing FLASH
NET "T_PROM_ENABLE_o" LOC = "P62";  # drive high to save power on EEPROM

# The below signal is to be used if you need a clock other than
#   the 50 MHz clock.  Comment this out if you are not using an
#   additional clock.
#NET "s_clock_X2" PERIOD = 160;

# Signals to/from X2 - this is specifically for the X2
#   experimental design, which is AUX<0> to AUX<41>, and
#   these pins must match the experiment.ucf file, not by
#   pin number, but by X1_X2_AUX<#>
```

```
NET "DATA_FROM_X2_MULTCHK_i<0>" LOC = "p153";  # X1_X2_AUX<0>
NET "DATA_FROM_X2_MULTCHK_i<1>" LOC = "p151";  # X1_X2_AUX<1>
NET "DATA_FROM_X2_MULTCHK_i<2>" LOC = "p150";  # X1_X2_AUX<2>
NET "DATA_FROM_X2_CNTCHK_i<0>" LOC = "p149";  # X1_X2_AUX<3>
NET "DATA_FROM_X2_CNTCHK_i<1>" LOC = "p147";  # X1_X2_AUX<4>
NET "DATA_FROM_X2_CNTCHK_i<2>" LOC = "p146";  # X1_X2_AUX<5>
NET "DATA_TO_X2_RESET_o" LOC = "p145";  # X1_X2_AUX<6>
#NET "DATA_FROM_X2_READY_i" LOC = "p144";  # X1_X2_AUX<7>
#NET "XXX" LOC = "p135";  # X1_X2_AUX<8>
#NET "XXX" LOC = "p134";  # X1_X2_AUX<9>
NET "DATA_FROM_X2_OUTPUT_i<0>" LOC = "p132";  # X1_X2_AUX<10>
NET "DATA_FROM_X2_OUTPUT_i<1>" LOC = "p127";  # X1_X2_AUX<11>
NET "DATA_FROM_X2_OUTPUT_i<2>" LOC = "p126";  # X1_X2_AUX<12>
NET "DATA_FROM_X2_OUTPUT_i<3>" LOC = "p120";  # X1_X2_AUX<13>
NET "DATA_FROM_X2_OUTPUT_i<4>" LOC = "p119";  # X1_X2_AUX<14>
NET "DATA_FROM_X2_OUTPUT_i<5>" LOC = "p112";  # X1_X2_AUX<15>
NET "DATA_FROM_X2_OUTPUT_i<6>" LOC = "p111";  # X1_X2_AUX<16>
NET "DATA_FROM_X2_OUTPUT_i<7>" LOC = "p110";  # X1_X2_AUX<17>
NET "DATA_FROM_X2_OUTPUT_i<8>" LOC = "p109";  # X1_X2_AUX<18>
NET "DATA_FROM_X2_OUTPUT_i<9>" LOC = "p108";  # X1_X2_AUX<19>
NET "DATA_FROM_X2_OUTPUT_i<10>" LOC = "p107";  # X1_X2_AUX<20>
NET "DATA_FROM_X2_OUTPUT_i<11>" LOC = "p105";  # X1_X2_AUX<21>
NET "DATA_FROM_X2_OUTPUT_i<12>" LOC = "p104";  # X1_X2_AUX<22>
NET "DATA_FROM_X2_OUTPUT_i<13>" LOC = "p103";  # X1_X2_AUX<23>
NET "DATA_FROM_X2_OUTPUT_i<14>" LOC = "p102";  # X1_X2_AUX<24>
NET "DATA_FROM_X2_OUTPUT_i<15>" LOC = "p101";  # X1_X2_AUX<25>
NET "DATA_FROM_X2_OUTPUT_i<16>" LOC = "p98";  # X1_X2_AUX<26>
NET "DATA_FROM_X2_OUTPUT_i<17>" LOC = "p97";  # X1_X2_AUX<27>
NET "DATA_FROM_X2_OUTPUT_i<18>" LOC = "p96";  # X1_X2_AUX<28>
NET "DATA_FROM_X2_OUTPUT_i<19>" LOC = "p94";  # X1_X2_AUX<29>
NET "DATA_FROM_X2_OUTPUT_i<20>" LOC = "p93";  # X1_X2_AUX<30>
NET "DATA_FROM_X2_OUTPUT_i<21>" LOC = "p92";  # X1_X2_AUX<31>
NET "DATA_FROM_X2_OUTPUT_i<22>" LOC = "p91";  # X1_X2_AUX<32>
NET "DATA_FROM_X2_OUTPUT_i<23>" LOC = "p90";  # X1_X2_AUX<33>
NET "DATA_FROM_X2_OUTPUT_i<24>" LOC = "p89";  # X1_X2_AUX<34>
NET "DATA_FROM_X2_OUTPUT_i<25>" LOC = "p88";  # X1_X2_AUX<35>
NET "DATA_FROM_X2_OUTPUT_i<26>" LOC = "p82";  # X1_X2_AUX<36>
NET "DATA_FROM_X2_OUTPUT_i<27>" LOC = "p81";  # X1_X2_AUX<37>
NET "DATA_FROM_X2_OUTPUT_i<28>" LOC = "p80";  # X1_X2_AUX<38>
NET "DATA_FROM_X2_OUTPUT_i<29>" LOC = "p79";  # X1_X2_AUX<39>
NET "DATA_FROM_X2_OUTPUT_i<30>" LOC = "p78";  # X1_X2_AUX<40>
NET "DATA_FROM_X2_OUTPUT_i<31>" LOC = "p77";  # X1_X2_AUX<41>
#NET "XXX" LOC = "p75"; # X1_X2_AUX<42> -- available on Flight Board
#NET "XXX" LOC = "p74"; # X1_X2_AUX<43> -- not avail on Flight Board
#NET "XXX" LOC = "p71"; # X1_X2_AUX<44> -- not avail on Flight Board

# X1/X2 Aux 43, and 44 are NOT available on the Flight Board
#   Aux 42 is still available on the Flight Board if needed.
#**********************************************************
# END signals to/from X2
#**********************************************************

#Flash Interface Signals
NET "T_FLASH_DATA_i<0>" LOC = "P207";
NET "T_FLASH_DATA_i<1>" LOC = "P209";
NET "T_FLASH_DATA_i<2>" LOC = "P212";
```

```
NET "T_FLASH_DATA_i<3>" LOC = "P216";
NET "T_FLASH_DATA_i<4>" LOC = "P218";
NET "T_FLASH_DATA_i<5>" LOC = "P220";
NET "T_FLASH_DATA_i<6>" LOC = "P223";
NET "T_FLASH_DATA_i<7>" LOC = "P226";
NET "T_FLASH_DATA_i<8>" LOC = "P208";
NET "T_FLASH_DATA_i<9>" LOC = "P211";
NET "T_FLASH_DATA_i<10>" LOC = "P213";
NET "T_FLASH_DATA_i<11>" LOC = "P217";
NET "T_FLASH_DATA_i<12>" LOC = "P219";
NET "T_FLASH_DATA_i<13>" LOC = "P222";
NET "T_FLASH_DATA_i<14>" LOC = "P224";
NET "T_FLASH_DATA_i<15>" LOC = "P225";
NET "T_FLASH_ADDRESS_o<0>" LOC = "P206";
NET "T_FLASH_ADDRESS_o<1>" LOC = "P205";
NET "T_FLASH_ADDRESS_o<2>" LOC = "P204";
NET "T_FLASH_ADDRESS_o<3>" LOC = "P198";
NET "T_FLASH_ADDRESS_o<4>" LOC = "P197";
NET "T_FLASH_ADDRESS_o<5>" LOC = "P196";
NET "T_FLASH_ADDRESS_o<6>" LOC = "P195";
NET "T_FLASH_ADDRESS_o<7>" LOC = "P194";
NET "T_FLASH_ADDRESS_o<8>" LOC = "P182";
NET "T_FLASH_ADDRESS_o<9>" LOC = "P183";
NET "T_FLASH_ADDRESS_o<10>" LOC = "P184";
NET "T_FLASH_ADDRESS_o<11>" LOC = "P185";
NET "T_FLASH_ADDRESS_o<12>" LOC = "P188";
NET "T_FLASH_ADDRESS_o<13>" LOC = "P189";
NET "T_FLASH_ADDRESS_o<14>" LOC = "P190";
NET "T_FLASH_ADDRESS_o<15>" LOC = "P192";
NET "T_FLASH_ADDRESS_o<16>" LOC = "P193";
NET "T_FLASH_ADDRESS_o<17>" LOC = "P177";
NET "T_FLASH_ADDRESS_o<18>" LOC = "P178";
NET "T_FLASH_ADDRESS_o<19>" LOC = "P179";
NET "T_FLASH_ADDRESS_o<20>" LOC = "P181";
NET "T_FLASH_WE_o" LOC = "P165";
NET "T_FLASH_RP_o" LOC = "P166";
NET "T_FLASH_WP_o" LOC = "P167";
NET "T_FLASH_CE_A_o" LOC = "P164";
#NET "T_Flash_CE_B_o" LOC = "P125"; # doesn't do anything!
NET "T_FLASH_OE_o" LOC = "P162";


#PC/104 Interface Signals
NET "T_Data_io<0>" LOC = "P11";          #ISA Data Bit 0  p. 11
NET "T_Data_io<1>" LOC = "P10";                  #ISA Data Bit 1  p. 10
NET "T_Data_io<2>" LOC = "P9";           #ISA Data Bit 2      p. 09
NET "T_Data_io<3>" LOC = "P7";           #ISA Data Bit 3  p. 07
NET "T_Data_io<4>" LOC = "P6";           #ISA Data Bit 4  p. 06
NET "T_Data_io<5>" LOC = "P5";           #ISA Data Bit 5  p. 05
NET "T_Data_io<6>" LOC = "P4";           #ISA Data Bit 6  p. 04
NET "T_Data_io<7>" LOC = "P3";           #ISA Data Bit 7  p. 03
NET "T_Address_i<0>" LOC = "P47";        #ISA Address 0        p. 47
NET "T_Address_i<1>" LOC = "P46";        #ISA Address 1        p. 46
NET "T_Address_i<2>" LOC = "P45";        #ISA Address 2        p. 45
NET "T_Address_i<3>" LOC = "P39";        #ISA Address 3        p. 39
NET "T_Address_i<4>" LOC = "P36";        #ISA Address 4        p. 36
NET "T_Address_i<5>" LOC = "P34";        #ISA Address 5        p. 34
```

```
NET "T_Address_i<6>" LOC = "P32";              #ISA Address 6           p. 32
NET "T_Address_i<7>" LOC = "P29";              #ISA Address 7           p. 29
NET "T_Address_i<8>" LOC = "P25";              #ISA Address 8           p. 25
NET "T_Address_i<9>" LOC = "P23";              #ISA Address 9           p. 23
#NET "T_Address_i<10> LOC = "p21";    #ISA Address 10    p. 21
NET "T_IORead_i" LOC = "P19";                  #CARD_OE                 p. 19
NET "T_IOWRITE_i" LOC = "P17";                 #CARD_WE            p. 17
NET "T_IOCS_i" LOC = "P16";              #CARD_CS3, used to be p. 13
NET "T_INTRPT_o" LOC = "P54";  # T_INTRPT_o is "p43" on the flight board
                  # This is ISA Interrupt 0, IRQ 7
           # NOTE: on the development board, IO Pin 5(P54)
           #  is jumpered to card interrupt 0
           #  actually is interrupt 6 on ARM side!


# SelectMap interface signals
NET "T_CCLK_o" LOC = "P69";  #Drive X2's CCLK pin, flight_board = p65
NET "T_SELECTMAP_INIT_o" LOC = "P117";      #Drive X2's INIT pin, flight_board = p124
                  # schematic says 124 = "IO_VREF_3"  ???
NET "T_SELECTMAP_WRITE_o" LOC = "P176";  #Drive X2's WRITE pin, flight_board = p63
                  # schematic says 63 = "IO_VREF_5"   ???
NET "T_SELECTMAP_CS_o" LOC = "P175";       #Drive X2's CS pin, flight_board = p64


# MLS swap pins so D(0) is LSB
NET "T_SELECTMAP_DATA_io<7>" LOC = "P169"; # X2_D0, flight_board = p68
NET "T_SELECTMAP_DATA_io<6>" LOC = "P128"; # X2_D1, flight_board = p69
NET "T_SELECTMAP_DATA_io<5>" LOC = "P131"; # X2_D2, flight_board = p70
NET "T_SELECTMAP_DATA_io<4>" LOC = "P137"; # X2_D3, flight_board = p71
NET "T_SELECTMAP_DATA_io<3>" LOC = "P148"; # X2_D4, flight_board = p74
NET "T_SELECTMAP_DATA_io<2>" LOC = "P155"; # X2_D5, flight_board = p75
NET "T_SELECTMAP_DATA_io<1>" LOC = "P158"; # X2_D6, flight_board = p121
NET "T_SELECTMAP_DATA_io<0>" LOC = "P168"; # X2_D7, flight_board = p122
# NET "T_SELECTMAP_BUSY_i" LOC = "P118";  #not sure what this does


NET "T_clock_i" LOC = "P87"; # Flight_board = P87
                  # P87 is a 51 MHz CARD_BCLK from ARM board
                  #    IF used on the development board
                  # For the flight board, P199 is unconnected.
                  # P87 has to be used on the flight board.
                  # If using P87 on the development board, then
                  #    ensure P199 is used on X2 for dev_board.
                  # For the flight board, X2's clock MUST be
                  #    tied to P199, and X1's clock to P87.


NET "T_X2_MODE<0>" LOC = "P160";
NET "T_X2_MODE<1>" LOC = "P159";
NET "T_X2_MODE<2>" LOC = "P161";
NET "T_X2_PROG_o" LOC = "P49";
#NET "T_clock_i" PERIOD = 20;
#NET "s_clock" PERIOD = 20;
```

X2 CONSTRAINT FILE

   This is the code listing for X2's constraint file in support of the TMR Multiplier. Pay particular attention to the blue comments next to the pin assignments, and compare these to X1's constraint file and corresponding pin assignments.

```
# Pin assignments for X2 (TMR Multiplier)
# by Jerry Caldwell
#
#   Double-check all pin assignments!
#
# All pin assignments in the comments following the actual pin
# locations must match same commented locations on control.ucf
# Example:  "p132" on tmr_multiply.ucf matches "p153" on control.ucf

# system clock
NET "clock" LOC = "P199"; # use this one for the 51 MHz oscillator
#NET "clock" LOC = "P87"; # use this one for a 50 MHz clock
# NET "clock" PERIOD = 40;
# NET "s_clock" PERIOD = 80;

# signals to/from X1
NET "mult_check<0>" LOC = "p132";  # X1_X2_AUX<0>
NET "mult_check<1>" LOC = "p134";  # X1_X2_AUX<1>
NET "mult_check<2>" LOC = "p135";  # X1_X2_AUX<2>
NET "cnt_check<0>" LOC = "p136";  # X1_X2_AUX<3>
NET "cnt_check<1>" LOC = "p138";  # X1_X2_AUX<4>
NET "cnt_check<2>" LOC = "p139";  # X1_X2_AUX<5>
NET "x1_reset" LOC = "p141";  # X1_X2_AUX<6>
#NET "data_rdy" LOC = "p144";  # X1_X2_AUX<7>
#NET "XXX" LOC = "p146";  # X1_X2_AUX<8>
#NET "XXX" LOC = "p147";  # X1_X2_AUX<9>
NET "result<0>" LOC = "p153";  # X1_X2_AUX<10>
NET "result<1>" LOC = "p154";  # X1_X2_AUX<11>
NET "result<2>" LOC = "p159";  # X1_X2_AUX<12>
NET "result<3>" LOC = "p160";  # X1_X2_AUX<13>
NET "result<4>" LOC = "p161";  # X1_X2_AUX<14>
NET "result<5>" LOC = "p177";  # X1_X2_AUX<15>
NET "result<6>" LOC = "p178";  # X1_X2_AUX<16>
NET "result<7>" LOC = "p179";  # X1_X2_AUX<17>
NET "result<8>" LOC = "p181";  # X1_X2_AUX<18>
NET "result<9>" LOC = "p182";  # X1_X2_AUX<19>
NET "result<10>" LOC = "p183";  # X1_X2_AUX<20>
NET "result<11>" LOC = "p184";  # X1_X2_AUX<21>
NET "result<12>" LOC = "p185";  # X1_X2_AUX<22>
NET "result<13>" LOC = "p188";  # X1_X2_AUX<23>
NET "result<14>" LOC = "p189";  # X1_X2_AUX<24>
NET "result<15>" LOC = "p190";  # X1_X2_AUX<25>
NET "result<16>" LOC = "p192";  # X1_X2_AUX<26>
NET "result<17>" LOC = "p193";  # X1_X2_AUX<27>
NET "result<18>" LOC = "p194";  # X1_X2_AUX<28>
NET "result<19>" LOC = "p195";  # X1_X2_AUX<29>
```

NET "result<20>" LOC = "p196"; # X1_X2_AUX<30>
NET "result<21>" LOC = "p197"; # X1_X2_AUX<31>
NET "result<22>" LOC = "p198"; # X1_X2_AUX<32>
NET "result<23>" LOC = "p204"; # X1_X2_AUX<33>
NET "count<0>" LOC = "p205"; # X1_X2_AUX<34>
NET "count<1>" LOC = "p206"; # X1_X2_AUX<35>
NET "count<2>" LOC = "p207"; # X1_X2_AUX<36>
NET "count<3>" LOC = "p208"; # X1_X2_AUX<37>
NET "count<4>" LOC = "p209"; # X1_X2_AUX<38>
NET "count<5>" LOC = "p211"; # X1_X2_AUX<39>
NET "count<6>" LOC = "p212"; # X1_X2_AUX<40>
NET "count<7>" LOC = "p213"; # X1_X2_AUX<41>

NET "XXX" LOC = "p216"; # X1_X2_AUX<42> # available on dev board
NET "XXX" LOC = "p217"; # X1_X2_AUX<43> # available on dev board
NET "XXX" LOC = "p218"; # X1_X2_AUX<44> # available on dev board

X1_X2_AUX<42,43,44> NOT availabe on flight board
42 was replaced by CE_B for flash

CONTROL.UCF – FLIGHT BOARD

Included below is the section of the constraint file for X1 for the Flight Board that differs from the Development Board. The below section of code can be compared to the constraint file for Development Board listed previously. It is easy to recognize that the only differences are in the actual numbers assigned to the pins below. The specific signal names remain exactly the same.

NET "T_INTRPT_o" LOC = "P54"; # T_INTRPT_o is "p43" on the flight board
                # This is ISA Interrupt 0, IRQ 7
            # NOTE: on the development board, IO Pin 5(P54)
            # is jumpered to card interrupt 0
            # actually is interrupt 6 on ARM side!

Selectmap interface signals
NET "T_CCLK_o" LOC = "P69"; #Drive X2's CCLK pin, flight_board = p65
NET "T_SELECTMAP_INIT_o" LOC = "P117";   #Drive X2's INIT pin, flight_board = p124
                # schematic says 124 = "IO_VREF_3" ???
NET "T_SELECTMAP_WRITE_o" LOC = "P176"; #Drive X2's WRITE pin, flight_board = p63
                # schematic says 63 = "IO_VREF_5"   ???
NET "T_SELECTMAP_CS_o" LOC = "P175";   #Drive X2's CS pin, flight_board = p64

MLS swap pins so D(0) is LSB
NET "T_SELECTMAP_DATA_io<7>" LOC = "P169"; # X2_D0, flight_board = p68
NET "T_SELECTMAP_DATA_io<6>" LOC = "P128"; # X2_D1, flight_board = p69
NET "T_SELECTMAP_DATA_io<5>" LOC = "P131"; # X2_D2, flight_board = p70
NET "T_SELECTMAP_DATA_io<4>" LOC = "P137"; # X2_D3, flight_board = p71
NET "T_SELECTMAP_DATA_io<3>" LOC = "P148"; # X2_D4, flight_board = p74
NET "T_SELECTMAP_DATA_io<2>" LOC = "P155"; # X2_D5, flight_board = p75

97

NET "T_SELECTMAP_DATA_io<1>" LOC = "P158"; # X2_D6, flight_board = p121
NET "T_SELECTMAP_DATA_io<0>" LOC = "P168"; # X2_D7, flight_board = p122
NET "T_SELECTMAP_BUSY_i" LOC = "P118"; #not sure what this does

NET "T_clock_i" LOC = "P199"; # Flight_board = P87

MAKEFILE FOR THE CONTROLLER CODE

Below is only the first portion of the Makefile that is used to compile the Controller Code. The code highlighted in red is the only part that needs to be changed by prospective Designers. These changes are done to identify specifics experiments in the output data stream and are therefore important.


```
###Makefile for compiling VHDL code

####################################################################
###                    Paths                    ###
####################################################################
# Top level project name (used also in naming source/output files below
# You don't have to use projname as root of filenames (you can change
# them below if you'd like), but it
# makes it easy to reuse this Makefile for a different design
PROJNAME      = control
ENTITYNAME  = cftp_ARM
# ID is used by the rd.sh program to determine how the output from
# your code should be formatted.  It is any 2 digit string, tell
# Mindy what you chose and she will add it to the rd.sh program.
# Already taken:
# JS: Josh's Cordic
# JM: Jerry's Multiplier
# SR: James' Shift Register
# FD: Flash Dump
# VT: V2 Test code
# FE: Flash Erase
ID        = JM
DESCR     = "Jerry's Multiplier"
#Location of all local source files
# MUST BE FULL PATH, XST doesn't like relative paths
SRCPATHLOC  = $(PROJNAME)_src
```

APPENDIX C: DATA FORMATTING CODE

Programs specifically designed for viewing data output from the CFTP architecture have been written in "C" code, and are included for the benefit of future designers who desire formatted output, such as that located in Figures 6 – 8 and 10 and 11. To view output data in this format, three files must be slightly modified. One is a Makefile, and the other two are "C" code programs that perform the function of reading data and outputting it in a clear, organized format.

The majority of the modifications of these three files merely require copying a section of code and pasting it, then slightly modifying it specific to an experiments output stream. The portion that will be edited in the top level program, rd_top_arm.c, ties into the modification designers make to the file Makefile_control for the Controller code, described in Chapter V and Appendix B. Specifically, the program rd_top_arm.c looks for the two initials added to Makefile_control, and then will call another c-code program that directly formats the data.

The sections of the three files that require modification are highlighted in red, followed by a description of what the code does.

TOP LEVEL C-CODE PROGRAM - rd_top_arm.c

```
#include<sys/io.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<stdio.h>
#include<unistd.h>

int printusageandexit() {
    printf("Usage: rd_top_arm\n"
      "with options: \n"
      "-f FILENAME    full path to filename to read\n"
      "-s SPEED       SPEED times real time\n" );
    return 0;
}

main(int argc, char **argv)
{
    unsigned          char          byte1,byte2,dummy,string[]="PART          SUCCESSFULLY
PROGRAMMED",synch_word[]="PART SUCCESSFULLY PROGRAMMED";
    int synch_len,i;
    char *fn, *cp, *speed, *cfn;
    FILE *child, *datafile;
```

```c
        fn = "/home/msurratt/work/arm_code/test";
        speed = "1";

        for (i=1; i<argc; i++) {
           if (strcmp(argv[i], "-f") == 0) {
              if (++i <= argc-1) { fn=argv[i]; }
              else { return printusageandexit(); }
           }
           else if (strcmp(argv[i], "-s") == 0) {
              if (++i <= argc-1) { speed = argv[i]; }
              else { return printusageandexit(); }
           }
           else { return printusageandexit(); }
        }

        datafile = fopen(fn,"r");

        fflush(NULL);
        if ( (child = popen("cat","w")) < 0) {
           printf("Failed to open child process\n");
           exit(0);
        }

        synch_len = strlen(synch_word);

        while(1) {

           for (i=0;i<synch_len-1;i++) {
              string[i] = string[i+1];
           }

           fread(&string[synch_len-1],1,1,datafile);
           fflush(NULL);
           fprintf(child,"%c",string[synch_len-1]);
           fflush(NULL);

           if (strcmp (string,synch_word) == 0) {
              fread(&byte1,1,1,datafile); // 0x0a
              fread(&byte2,1,1,datafile); // 0x0d
              fprintf(child,"%c%c\n",byte1,byte2);
              fflush(NULL);
              fread(&byte1,1,1,datafile); // ID 1
              fread(&byte2,1,1,datafile); // ID 2
              fread(&dummy,1,1,datafile); // 0x0d
              pclose(child);
// for EXPERIMENTAL DESIGN
// add an entry for your unique id, and executable
              // Josh's cordic, ID JS
              if (byte1 == 'J' && byte2 == 'S') {
                 fflush(NULL);
                 if ( (child = popen("cordic_out","w")) < 0) {
                    fprintf(stderr,"Failed to open child process\n");
                    exit(0);
                 }
              }
```

100

```
        // Jerry's Dual Counter ID JD
        if (byte1 == 'J' && byte2 == 'D') {
          fflush(NULL);
          cp = "dualcount_out_arm ";
          if ((cfn = (char *) malloc(strlen(cp)+strlen(speed)+2)) == NULL ) { fprintf(stderr,"malloc
failed"); return 0; }
          strcpy(cfn,cp);
          strcat(cfn,speed);
          if ( (child = popen(cfn,"w")) < 0) {
            fprintf(stderr,"Failed to open child process\n");
            exit(0);
          }
        }
        // Jerry's Multiplier ID JM
        if (byte1 == 'J' && byte2 == 'M') {
          fflush(NULL);
          cp = "mult_out_arm ";
          if ((cfn = (char *) malloc(strlen(cp)+strlen(speed)+2)) == NULL ) { fprintf(stderr,"malloc
failed"); return 0; }
          strcpy(cfn,cp);
          strcat(cfn,speed);
          if ( (child = popen(cfn,"w")) < 0) {
            fprintf(stderr,"Failed to open child process\n");
            exit(0);
          }
        }
```

The above section beginning with the comment "//Jerry's Multiplier ID JM," was copied and only two lines were modified. In the first "if" statement, "J" and "M" were inserted next to the double-equal symbols. This tells the rd_top_arm program to look for these two initials in a given file of data, which are included in that data because these two initials were added to the file Makefile_control. Two lines below that, mult_out_arm was inserted in between the quotation marks. This tells the rd_top_arm program to call the specific c-code program that reads the multiplier output, named appropriately.

```
        // Mindy's Counter ID MS
        if (byte1 == 'M' && byte2 == 'S') {
          fflush(NULL);
          cp = "mindy_out ";
          if ((cfn = (char *) malloc(strlen(cp)+strlen(speed)+2)) == NULL ) { fprintf(stderr,"malloc
failed"); return 0; }
          strcpy(cfn,cp);
          strcat(cfn,speed);
          if ( (child = popen(cfn,"w")) < 0) {
            fprintf(stderr,"Failed to open child process\n");
            exit(0);
          }
        }
        // Josh's cordic with timestamp (for flight), ID JA
        if (byte1 == 'J' && byte2 == 'A') {
```

```
        fflush(NULL);
        cp = "cordic_out_arm ";
        if ((cfn = (char *) malloc(strlen(cp)+strlen(speed)+2)) == NULL ) { fprintf(stderr,"malloc
failed"); return 0; }
        strcpy(cfn,cp);
        strcat(cfn,speed);
        if ( (child = popen(cfn,"w")) < 0) {
           fprintf(stderr,"Failed to open child process\n");
           exit(0);
        }
      }
    }
  }
}
```

SPECIFIC C-CODE PROGRAM – "name"_out_arm.c

The file included below is named mult_out_arm.c, and is the file that
rd_top_arm.c will call when rd_top_arm.c encounters a "J" and an "M" in an output file.
The portion in red is the area that requires modification if data is to be organized into an
easily readable format.

```
//mult_out_arm.c modified by Jerry Caldwell
//21 Sept 06
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include "rd_sub.h"
#include "smrb_locs.h"

int getER();
void getPC();

int main(int argc, char **argv){

   unsigned char synch_word[]="AA";
   unsigned char string[]="PART SUCCESSFULLY PROGRAMMED";
   long timestamp,oldTime = 0;
   int speed;
        int total_SM=0;
   smrb_locs locs;

   locs.wrptr = 0;
   locs.end_of_array = 0;
   locs.data = NULL;

   speed = atoi(argv[1]);

   while (1) {
```

```
            synch_word[0] = synch_word[1];
            fread(&synch_word[1],1,1,stdin);

            if ( new_output(&string[0],synch_word[1]) ) {
                return 0;
            }
            if (strcmp(synch_word,"ER") == 0 ) {
                if((timestamp = getER(oldTime,speed)) == 0 ) {
                    fprintf(stderr,"getER failed");
                }
                oldTime = timestamp;
            }
            else if (strcmp(synch_word,"SM") == 0 ) {
                oldTime = getTS();
                printf("\ntimestamp: %08x ",oldTime);
                fflush(NULL);
                // return last read value: beginning of SC or ER
                if( ( synch_word[1] = getSM(&locs,total_SM) ) == 0) {
                    fprintf(stderr,"getSM failed");
                }
                SMRC(&locs);
                                total_SM++;
            }
            else if (strcmp(synch_word,"SC") == 0 ) {
                oldTime = getTS();
                printf("timestamp: %08x \n",oldTime);
                SMRC(&locs);
            }
            else if (strcmp(synch_word,"PC") == 0 ) {
                                getPC();
                SMRC(&locs);
            }
        }
    }
}
void getPC() {

    unsigned char c;
    fread(&c,1,1,stdin); //BLOCK NUM
    printf("BLK#: %02x   ",c);
    fread(&c,1,1,stdin); //MJA
    printf("MJA#: %02x   ",c);
    fread(&c,1,1,stdin); //MNA
    printf("MNA#: %02x   ",c);
    fread(&c,1,1,stdin); //BIT Upper
    printf("BIT#: %02x",c);
    fread(&c,1,1,stdin); //BIT Lower
    printf("%02x   ",c);
    fread(&c,1,1,stdin); //FLASH Offset Upper
    printf("FLASH OFFSET: %02x",c);
    fread(&c,1,1,stdin); //FLASH Offset Middle
    printf("%02x",c);
    fread(&c,1,1,stdin); //FLASH Offset Lower
    printf("%02x\n",c);
}
int getER(long oldTime, int speed) {
```

```
    unsigned char c;
    long timestamp;

// Use the below portion to read the TMR multipliers
    fread(&c,1,1,stdin);  // this reads the padded zeros before
                  // err_cnt, if those are output from the X1 code
    fread(&c,1,1,stdin);
    printf(" Error Count: ");
    printf("%02x  ",c);
    printf(" ");

    printf("Count_voter: ");
    fread(&c,1,1,stdin);
    printf("%02x  ",c);
    printf(" ");

    printf("Mult_voter: ");
    fread(&c,1,1,stdin);
    printf("%02x  ",c);
    printf(" ");

    printf("Count: ");
    fread(&c,1,1,stdin);
    printf("%02x  ",c);
    printf(" ");

    printf("Count Squared: ");
    fread(&c,1,1,stdin);
    printf("%02x",c);
    fread(&c,1,1,stdin);
    printf("%02x",c);
    fread(&c,1,1,stdin);
    printf("%02x  ",c);
    printf(" ");
```

The above portion of code uses "fread" and "printf" to read bytes of data and print it next to the strings in quotation marks. It should be self evident how the strings in quotation marks are altered to identify specific data. For each byte of data that is read, a corresponding "printf" statement is needed to print that data. The location of the "printf" statement determines the order in which the data will be printed. Notice the uses of quotation marks to format the bytes of data as well as spaces to provide additional formatting.

```
  printf("Timestamp: ");
   timestamp = getTS();
   printf("%08x\n",timestamp);
   fflush(NULL);

   if (timestamp == 0) { timestamp = 1; }
```

```
    return(timestamp);
}
```

MAKEFILE

This file must be modified to so that rd_top_arm.c and the specific c-file created, mult_out_arm.c for this particular example, can be compiled. The portions in red are the lines of code requiring modification.

```
INC=./include
FLASH_SRC=./mkflash_src
RD_SRC=./process_output_src
INJERR_SRC=./inject_error_src
BIN=.

all: rd_top_arm mult_out_arm count_out_arm dualcount_out_arm
```

The only change made to the above line was to add the filename mult_out_arm.

```
inject_error: $(INJERR_SRC)/inject_error.c
        gcc -I$(INC) -o $(BIN)/inject_error $(INJERR_SRC)/inject_error.c

interleave_files: $(FLASH_SRC)/interleave_files.c
        gcc -I$(INC) -o $(BIN)/interleave_files $(FLASH_SRC)/interleave_files.c

strip_mask: $(FLASH_SRC)/strip_mask.c
        gcc -I$(INC) -o $(BIN)/strip_mask $(FLASH_SRC)/strip_mask.c

mult_out_arm: $(RD_SRC)/mult_out_arm.c $(RD_SRC)/smrb.c $(RD_SRC)/smrc.c
$(RD_SRC)/new_output.c $(RD_SRC)/smrb_locs.c $(INC)/rd_sub.h $(RD_SRC)/getTS.c
        gcc -I$(INC) -o $(BIN)/mult_out_arm $(RD_SRC)/mult_out_arm.c $(RD_SRC)/smrb.c
$(RD_SRC)/smrc.c $(RD_SRC)/new_output.c $(RD_SRC)/smrb_locs.c $(RD_SRC)/getTS.c -lm
```

The above portion of code was merely copied and pasted, and "mult_out_arm" and "mult_out_arm.c" was substituted in the appropriate places. Studying the code below makes is evident where these substitutions take place.

```
count_out_arm: $(RD_SRC)/count_out_arm.c $(RD_SRC)/smrb.c $(RD_SRC)/smrc.c
$(RD_SRC)/new_output.c $(RD_SRC)/smrb_locs.c $(INC)/rd_sub.h $(RD_SRC)/getTS.c
        gcc -I$(INC) -o $(BIN)/count_out_arm $(RD_SRC)/count_out_arm.c $(RD_SRC)/smrb.c
$(RD_SRC)/smrc.c $(RD_SRC)/new_output.c $(RD_SRC)/smrb_locs.c $(RD_SRC)/getTS.c -lm

dualcount_out_arm: $(RD_SRC)/dualcount_out_arm.c $(RD_SRC)/smrb.c $(RD_SRC)/smrc.c
$(RD_SRC)/new_output.c $(RD_SRC)/smrb_locs.c $(INC)/rd_sub.h $(RD_SRC)/getTS.c
        gcc -I$(INC) -o $(BIN)/dualcount_out_arm $(RD_SRC)/dualcount_out_arm.c $(RD_SRC)/smrb.c
$(RD_SRC)/smrc.c $(RD_SRC)/new_output.c $(RD_SRC)/smrb_locs.c $(RD_SRC)/getTS.c -lm
```

cordic_out_arm: $(RD_SRC)/cordic_out_arm.c $(RD_SRC)/smrb.c $(RD_SRC)/smrc.c
$(RD_SRC)/new_output.c $(RD_SRC)/smrb_locs.c $(INC)/rd_sub.h $(RD_SRC)/getTS.c
       gcc -I$(INC) -o $(BIN)/cordic_out_arm $(RD_SRC)/cordic_out_arm.c $(RD_SRC)/smrb.c
$(RD_SRC)/smrc.c $(RD_SRC)/new_output.c $(RD_SRC)/smrb_locs.c $(RD_SRC)/getTS.c -lm

sr_out: sr_out.c smrb.c smrc.c new_output.c smrb_locs.c rd_sub.h
       gcc -o $(BIN)/sr_out $(RD_SRC)/sr_out.c $(RD_SRC)/smrb.c $(RD_SRC)/smrc.c
$(RD_SRC)/new_output.c $(RD_SRC)/smrb_locs.c -lm

sr_out_c2: $(RD_SRC)/sr_out.c $(RD_SRC)/smrb_c2.c $(RD_SRC)/smrc.c $(RD_SRC)/new_output.c
$(RD_SRC)/smrb_locs.c $(RD_SRC)/rd_sub.h
       gcc -o $(BIN)/sr_out_c2 $(RD_SRC)/sr_out.c $(RD_SRC)/smrb_c2.c $(RD_SRC)/smrc.c
$(RD_SRC)/new_output.c $(RD_SRC)/smrb_locs.c -lm

rd_top: $(RD_SRC)/rd_top.c
       gcc -o $(BIN)/rd_top -g $(RD_SRC)/rd_top.c

rd_top_arm: $(RD_SRC)/rd_top_arm.c
       gcc -o $(BIN)/rd_top_arm -g $(RD_SRC)/rd_top_arm.c

COMMAND –LINE ENTRIES

Examples of the specific command-line entries to compile the c-code, and to read and print out data from a file, are included.

To compile the code, you must be in the same directory where the make file exists. Then the command-line entry is as simple as typing "make," as below.

cftp:~/directory$ make

To read the contents of a file and print them to a screen using these programs, use the command "rd_top_arm" followed by a "-f" denoting a file name to follow, then the exact name of the file to read.

cftp:~/directory$ rd_top_arm –f filename

LIST OF REFERENCES

1. Surratt, Mindy, "CFTP Development Environment Technical Manual", Naval Postgraduate School, Monterey California, April 2006.

2. Majewicz, Peter J., "Implementation of a Configurable Fault Tolerant Processor (CFTP) Using Internal Triple Modular Redundancy (TMR)," Master's Thesis, Naval Postgraduate School, December 2005.

3. Coudeyras, James C., "Radiation Testing of the Configurable Fault Tolerant Processor (CFTP) For Space-Based Applications," Master's Thesis, Naval Postgraduate School, Monterey, California, December 2005.

4. Ebert, Dean A., "Design and Development of a Configurable Fault Tolerant Processor (CFTP)," Master's Thesis, Naval Postgraduate School, Monterey, California, March 2003.

5. "PC/104 Specification Version 2.4," The PC/104 Embedded Consortium, San Jose, California, August 2001.

6. "QPro Virtex 2.5V Radiation Hardened FPGAs", Xilinx Preliminary Product Specification, DS028(v1.2), November 5, 2001.

7. "Virtex FPGA Series Configuration and Readback", Xilinx XAPP138 (v2.8), March 11, 2005.

8. "Using a Microprocessor to Configure Xilinx FGPAs via Slave Serial or SelectMap Mode", Xilinx XAPP502 (v1.4), March 11, 2005.

9. Snodgrass, Joshua D., "Low-Power Fault Tolerance for Spacecraft FPGA-Based Numerical Computing," PhD Dissertation, Naval Postgraduate School, September 2006.

10. "Project Navigator, Release Version: 6.3.03i," Copyright © 1995-2004 Xilinx, Inc.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1.   Defense Technical Information Center
        Ft. Belvoir, Virginia

2.   Dudley Knox Library
        Naval Postgraduate School
        Monterey, California

3.   Chairman, ECE Department, Knorr
        Naval Postgraduate School
        Monterey, California

4.   Marine Corps Representative
        Naval Postgraduate School
        Monterey, California

5.   Director, Training and Education, MCCDC, Code C46
        Quantico, Virginia

6.   Director, Marine Corps Research Center, MCCDC, Code C40RC
        Quantico, Virginia

7.   Professor Herschel H. Loomis
        Naval Postgraduate School
        Monterey, California

8.   Professor Alan A. Ross
        Naval Postgraduate School
        Monterey, California