NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

FAULT TOLERANT MICROCONTROLLER FOR THE CONFIGURABLE FAULT TOLERANT PROCESSOR

by

David Dwiggins Jr.

September 2008

Thesis Co-Advisors: Herschel H. Loomis Jr.
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** September 2008	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE Fault Tolerant Microcontroller for the Configurable Fault Tolerant Processor	**5. FUNDING NUMBERS**
6. AUTHOR(S) David E. Dwiggins Jr.	
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)

In this thesis, the design of a fault tolerant microcontroller for the Configurable Fault Tolerant Processor is presented. The Configurable Fault Tolerant processor is a spaceborne Field Programmable Gate Array experiment platform susceptible to Single Event Upsets. Fault tolerance is needed to control the experiment in higher radiation orbits and the microcontroller will offer enhanced functionality for experiments. The 16-bit microcontroller is contained within the resources of a single Field Programmable Gate Array. It includes RAM, microprocessor, FPGA configuration and configuration scrubbing modules, PC/104 interface module, and fault detection and correction capabilities. Fault tolerance is implemented via triple modular redundancy and Hamming error correction coding. Complete source code for the microcontroller and C-based compilation tools are included as appendices.

14. SUBJECT TERMS Single Event Upset (SEU), Field Programmable Gate Array (FPGA), Configurable Fault Tolerant Processor (CFTP), Triple Modular Redundancy (TMR)	**15. NUMBER OF PAGES** 225
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

FAULT TOLERANT MICROCONTROLLER FOR THE CONFIGURABLE FAULT TOLERANT PROCESSOR

David E. Dwiggins Jr.
Lieutenant, United States Navy
B.S., Carnegie Mellon University, 1993

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2008

Author: David E. Dwiggins Jr.

Approved by: Herschel H. Loomis Jr.
Thesis Co-Advisor

Alan A. Ross
Thesis Co-Advisor

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

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

In this thesis, the design of a fault tolerant microcontroller for the Configurable Fault Tolerant Processor is presented. The Configurable Fault Tolerant processor is a spaceborne Field Programmable Gate Array experiment platform susceptible to Single Event Upsets. Fault tolerance is needed to control the experiment in higher radiation orbits and the microcontroller will offer enhanced functionality for experiments. The 16-bit microcontroller is contained within the resources of a single Field Programmable Gate Array. It includes RAM, microprocessor, FPGA configuration and configuration readback modules, PC/104 interface module, and fault detection and correction capabilities. Fault tolerance is implemented via triple modular redundancy and Hamming error correction coding. Complete source code for the microcontroller and C-based compilation tools are included as appendices.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION....1
A. CFTP OBJECTIVE....1
B. RESEARCH OBJECTIVES....2
C. OVERVIEW....2
II. CFTP HARDWARE....5
A. PC/104 COMPUTER....5
B. EXPERIMENT BOARDS....5
1. Control FPGA....6
2. Experiment FPGA....7
3. XQR18V04 Configuration Flash PROM....7
4. Intel 28F320C3 32Mbit Flash RAM....7
5. 6 Elpida 256Mbit PC133 SDRAMs....8
C. SUMMARY....8
III. MICROCONTROLLER DESIGN....9
A. OBJECTIVE....9
B. FUNCTIONAL BLOCKS....10
1. PC/104 Interface Module....11
2. Timestamp Counter....12
3. SelectMap Configuration Module....12
4. SelectMap Readback Module....13
5. X2 Interface....13
C. SUMMARY....13
IV. PROCESSOR SELECTION AND FEATURES....15
A. SELECTING A PROCESSOR....15
B. XSOC GENERAL DESCRIPTION....15
C. SUMMARY....17
V. IMPLEMENTING FAULT TOLERANCE....19
A. TRIPLE MODULAR REDUNDANCY....19
B. XILINX ISE PARTITIONS....22
C. COMPLETE FAULT TOLERANCE....23
D. SUMMARY....23
VI. MODIFYING XSOC FOR CFTP....25
VII. MEMORY SUBSYSTEM....27
A. DESIGN....27
B. XSOC MEMORY MODIFICATIONS....27
C. MEMORY READ CYCLE....28
D. MEMORY WRITE CYCLE....30
E. ERROR CORRECTION CODING....30
F. ALTERNATIVE MEMORY OPTION....32

G. MEMORY SUBSYSTEM FILES....34
VIII. ADDING I/O MODULES....35
A. TOP-LEVEL INTERFACE....35
B. WRAPPER MODULE....36
C. I/O MODULE....37
D. X2 I/O MODULES....39
E. SUMMARY....40
IX. X2 CONFIGURATION MODULE....41
A. GENERAL DESCRIPTION....41
B. XSOC INTERFACE....41
C. VHDL CODE....41
X. X2 CONFIGURATION READBACK MODULE....43
A. GENERAL DESCRIPTION....43
B. XSOC INTERFACE....43
C. VHDL CODE....44
XI. ERROR REPORTING SUBSYSTEM....45
A. GENERAL DESCRIPTION....45
B. SYNDROME REPORTING BUS....45
C. VOTER CONNECTION....45
D. QUEUE....46
E. XSOC INTERFACE....47
F. FAULT TOLERANCE IN THE ERROR REPORTING MODULE....47
XII. PC/104 INTERFACE MODULE....49
A. GENERAL DESCRIPTION....49
B. XSOC INTERFACE....49
C. QUEUE....50
D. PC/104 INTERFACE....51
1. Data Reads....51
2. Data Writes....52
E. FAULT TOLERANCE IN THE PC/104 INTERFACE....53
XIII. TIMESTAMP COUNTER/INTERRUPT CONTROLLER MODULE....55
A. GENERAL DESCRIPTION....55
B. TIMESTAMP COUNTER....55
C. INTERRUPT CONTROLLER....56
D. TIMER....59
XIV. COMPILING/LOADING XSOC SOFTWARE....61
A. PREPARING FOR SOFTWARE DEVELOPMENT....61
B. SOFTWARE PROCESS EXAMPLE....63
XV. CONCLUSIONS AND RECOMMENDATIONS....65
A. SUMMARY....65
B. CONCLUSIONS....66
C. RECOMMENDATIONS....67

1. Implement Configuration Scrubbing on the X1 FPGA..................67
2. Modify the Code to Eliminate "Keeper" Circuits..........................67
3. Make a "Mini-OS" for the CFTP Microcontroller........................67
4. Enable 16-bit Transfers on the PC/104 Bus....................................68
5. Automated Memory Scrubbing..68

APPENDIX A: VERILOG SOURCE CODE..69
A. BITS.V..69
B. BRAM4KX22.V...70
C. BRAM_ECC.V...74
D. CLOCKDIV.V..76
E. CTRL.V...77
F. DATAPATH.V..92
G. ERRORFIFO.V..100
H. IOTEMPLATE.V..103
I. MEMCTRL.V...105
J. MEMIO.V..113
K. PC104QUEUE.V..113
L. PCFILE.V..117
M. QUEUE1.V...117
N. REGISTER16X16.V...120
O. SELECTMAP_CONFIG_XSOC.V...122
P. SELECTMAP_RB_XSOC.V...127
Q. TMRND.V...134
R. VOTERIDS.V..137
S. X1CONTROL.V..140
T. XR16.V..146
U. XSCOUNTER.V...154
V. XSOC.V...162
W. XSOC_PC104.V..167

APPENDIX B: VHDL SOURCE CODE...177
A. SELECTMAP_CONFIG.VHD...177
B. SELECTMAP_READBACK.VHD..180

APPENDIX C: C SOURCE CODE..189
A. HEXTOV.C..189
B. LIBXR16.C..195

APPENDIX D: ASSEMBLY FILES...197
A. RESET.S..197

APPENDIX E: XSOC LICENSE AGREEMENT..199

LIST OF REFERENCES...203

INITIAL DISTRIBUTION LIST...205

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1. CFTP Flight Board Photograph 6
Figure 2. CFTP Block Diagram – Before Microcontroller 10
Figure 3. CFTP Block Diagram – After Microcontroller..11
Figure 4. XSOC Data Path 16
Figure 5. Pipelined Execution. From [3]..17
Figure 6. Voter Strategy 20
Figure 7. Voter Operation 21
Figure 8. Replacement Register File 26
Figure 9. Datapath Outputs to Memory..28
Figure 10. Memory Data Flow 29
Figure 11. Fault Tolerant Memory 32
Figure 12. Alternative Memory Data Flow 33
Figure 13. Alternative Design for Fault Tolerant Memory 34
Figure 14. Interrupt Controller States...58

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. XQVR-600 FPGA Specifications [4] 7
Table 2. Parity Group Bits 31
Table 3. Typical IO Wrapper Module Ports 36
Table 4. Typical IO Module Ports 38
Table 5. Possible X2 I/O Interface Module 39
Table 6. X2 Configuration Module Address Map 41
Table 7. X2 Configuration Readback Module Address Map....43
Table 8. Syndrome Reporting Bus, Voter Format 45
Table 9. Queue Module Ports 46
Table 10. Error Reporting Module Address Map 47
Table 11. PC/104 Interface Address Map....49
Table 12. PC/104 Status Register 51
Table 13. PC/104 Interface Signals....52
Table 14. PC/104 Bus Address Map....52
Table 15. Counter Address Space....56
Table 16. Interrupt Controller Address Space 59
Table 17. Timer Address Space....60
Table 18. Timer Mask Values (20.4 MHz System Clock)....60

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Spaceborne computer hardware is faced with many challenges. One is the inability to alter hardware as mission needs change after initial system deployment. The use of Field Programmable Gate Arrays (FPGA)s offers an alternative to conventional fixed-function hardware and allows for reconfiguration in orbit. Another problem with computer hardware in space is the effect from cosmic radiation that causes Single Event Upsets (SEU)s. SEUs are routinely countered via fault tolerant techniques such as Triple Modular Redundancy (TMR) and Error Correction Coding (ECC). The reconfigurability of FPGAs comes with a cost; SEUs in FPGAs can cause not only data errors that would exist in conventional fixed function designs but can also affect configuration memory and could potentially alter hardware function.

The Configurable Fault Tolerant Processor (CFTP) was created in order to test the suitability of Field Programmable Gate Arrays (FPGA) in space applications. It contains two FPGAs and associated support hardware. One FPGA is designated the controller, X1, while the other is designated the experiment, X2. Besides implementing TMR and ECC techniques that one would use with fixed-function hardware, CFTP also implements configuration scrubbing. Configuration scrubbing is comparing the configuration of a FPGA with a stored copy and correcting the values found to be in error. CFTP is controlled by a PC/104 computer board which is then connected to the spacecraft computer for control and communication to the ground station. The FPGA controller design currently in use and the PC/104 computer are not fault tolerant and make CFTP unsuitable for deployment in high-radiation orbits.

The objective of this thesis was to re-implement the X1 control FPGA by adding fault tolerance and inserting a microcontroller to manage the fixed-function blocks. Fault tolerance was obtained through use of TMR and ECC wherever possible. The microcontroller on X1 offers more flexibility in operation of CFTP, including more complex functionality of X1 blocks. With dedicated functional blocks, changing the way

they cooperate requires complex hardware changes. With a microcontroller, simply changing the program offers the same results with a much faster development time.

This thesis describes the process of selecting the microcontroller, modifying it for fault tolerance, and changing its memory interface to best use the FPGA block RAMs. Two different memory options for the microcontroller are explored in detail. A software tool is created to load compiler output into the FPGA configuration. Additionally, error detection capabilities built into the TMR and ECC implementations are forwarded to a newly created error reporting system.

After describing the microcontroller, this thesis discusses the alterations to existing functional blocks for use with the microcontroller. An interrupt controller is constructed to enable more efficient interrupt-driven I/O. Detailed instructions for adding new I/O devices to the microcontroller are given. A step-by-step guide for programming and loading software into the microcontroller is presented. Finally, source code for creating the configuration for the X1 FPGA is included.

Early versions of the design were tested as an experiment on the CFTP hardware. The final version was also successfully tested on the CFTP hardware and with a software-based simulation tool. The result is more elaborate operations such as eliminating the PC/104 computer board and using X1 to format reports are now realizable.

ACKNOWLEDGMENTS

I wish to thank Professor Herschel H. Loomis Jr. and Professor Alan A. Ross for the endless hours they spent discussing and helping me prepare this thesis.

I also wish to thank the members of the CFTP Program for making CFTP the robust platform it is without which this thesis would not exist. Ron, Mindy, Rita and many others come to mind

I also wish to thank Jan Grey for providing an excellent System-on-a-Chip that formed a core component of this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. CFTP OBJECTIVE

Spaceborne systems are limited to the resources present at the time of initial system deployment. Conventional fixed-function computer hardware is subject to obsolescence and offers no capability for reconfiguration if system needs change. Another concern facing spaceborne systems is the effect of cosmic radiation resulting in Single-Event Upsets (SEU) that causes errors in computer results [1]. For this reason, properly designed spaceborne systems must be fault tolerant. Field Programmable Gate Arrays (FPGA) offer a solution for providing reconfigurable hardware but with a caveat: their configuration is stored in Random Access Memory (RAM), increasing their susceptibility to SEUs and requiring specialized techniques to counter them. The Configurable Fault Tolerant Processor (CFTP) is a system designed specifically for the purpose of on-orbit testing and evaluating the reliability of instantiated reliability-enhanced circuits in FPGAs.

CFTP is a project of the Space Systems Academic Group at the Naval Postgraduate School located in Monterey, California. It consists of two main assemblies: a PC/104 computer board and an experiment board. The PC/104 computer board is intended to control the experiment board and interface with the spacecraft for communication with the ground station.

The experiment board contains two Xilinx FPGAs and various other components. One FPGA is designated as a controller, X1, and the other is designated the experiment, X2. The control FPGA controls and configures the experiment FPGA as well as handling communication between the experiment FPGA and the PC/104 processor board. Even when the experiment FPGA contains fault tolerant experiments, the current X1 design and the PC/104 computer board are not fault tolerant.

A CFTP flight board was launched into a low earth orbit on March 9, 2007 aboard the United States Naval Academy's MidStar satellite [2]. CFTP functioned as designed and the platform was proven a success. The work done in this thesis is not yet implemented in the orbiting platform due to the lack of functional ground support to the MidStar satellite.

B. RESEARCH OBJECTIVES

The research objectives of this thesis were to redesign the processes running in the X1 control FPGA to include fault tolerance and to add a microcontroller to manage the functional blocks found within. The additional flexibility and functionality will allow for much more robust experiment implementations, while increased fault tolerance will allow for more reliable operation in higher radiation orbits. Finally, this thesis should serve as a reference manual for those using the revised X1 control system in future experiments.

C. OVERVIEW

Chapter II contains a brief description of CFTP hardware. Chapter III gives an idea of what changing to a microcontroller accomplished and how the existing hardware changed. Chapter IV discusses the selection of the processor used as a microcontroller and introduces the XSOC System-on-a-chip that was selected [3]. Chapter V details the methods used to implement fault tolerance, including the voter design for Triple Modular Redundancy (TMR). Chapter VI details the changes that were made to the XSOC SoC for use with CFTP. Chapter VII details the ECC memory subsystem that was designed for XSOC as well as a smaller-capacity TMR alternative that was not selected. Chapter VIII is a guide for how to add new I/O modules for use with the microcontroller. Chapters IX and X indicate how the existing X2 configuration and X2 configuration readback modules were integrated with the microcontroller. Chapter XI describes the error reporting subsystem and the global error-reporting bus. Chapter XII shows how the PC/104 interface was altered for use with the microcontroller. Chapter XIII describes the design of a module containing a timestamp counter, programmable timer and interrupt

controller. Chapter XIV contains information on how to compile and load software into the microcontroller. Chapter XV summarizes this thesis and suggests follow-on work.

Appendix A contains the Verilog HDL source code used to create the microcontroller and sample I/O module. Appendix B contains the VHDL source code for modules used in the microcontroller that were modified from the previous non-fault tolerant version of X1. Appendix C contains the C source code used to create a format conversion tool necessary for the software loading process and a modified version of the source code used to create the XSOC runtime library. Finally, Appendix E contains the XSOC License Agreement.

THIS PAGE INTENTIONALLY LEFT BLANK

II. CFTP HARDWARE

The CFTP hardware consists of two major assemblies: a PC/104 computer and the experiment board. They are connected via a 16-bit PC/104 bus.

A. PC/104 COMPUTER

Several different PC/104 computer boards have been used to control the various versions of the CFTP experiment. All use the Linux operating system and are connected via a PC/104 bus to the experiment board. All communication to and from the experiment board is done through the PC/104 computer. The PC/104 computer formats and forwards data to the spacecraft Command and Data Handler (CDH). Additionally, the PC/104 computer is responsible for loading configuration data into X1. The PC/104 computer is not fault tolerant. The PC/104 computer board is mounted beneath the CFTP board in the flight assembly shown in Figure 1.

B. EXPERIMENT BOARDS

There are several versions of experiment boards. The description below is specific to the flight board except where noted. A second version has a much larger experiment FPGA (a Virtex-II device) and another is constructed with standard non-radiation-hardened parts, but the control FPGA is the same basic model in all variants. Figure 1 shows a photograph of a CFTP flight board.

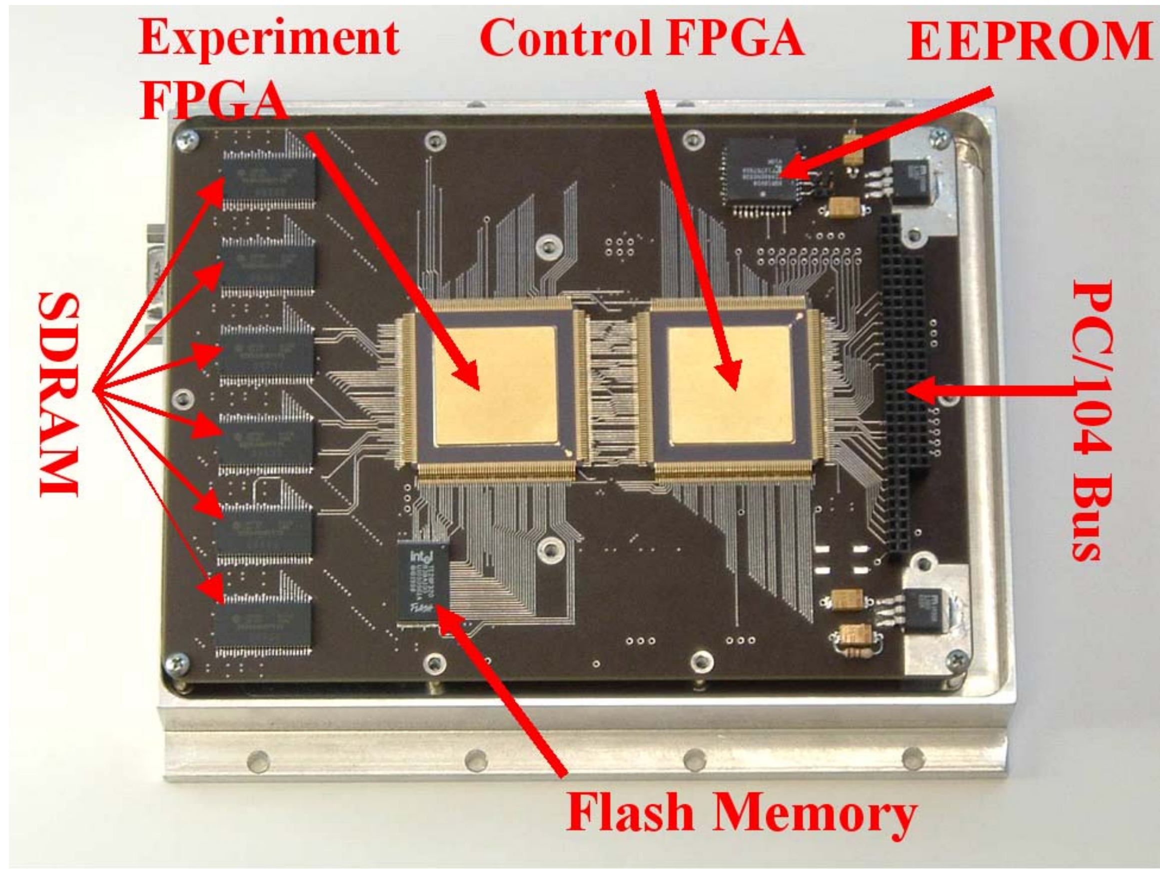

Figure 1. CFTP Flight Board Photograph

The flight board contains two Xilinx Virtex XQVR-600-4CB228 radiation-hardened FPGAs, a Xilinx QPro XQR18V04 radiation-tolerant 4Mbit configuration Programmable Read-Only Memory (PROM), Intel 28F320C3 32Mbit Flash RAM and six Elpida HM5225405B-75 256Mbit PC133 Synchronous Dynamic Random Access Memories (SDRAM).

1. Control FPGA

This FPGA, one of two Virtex XQVR-600s on the experiment board, is directly connected to the PC/104 interface, the XQR18V04 configuration Flash PROM, the 28F320C3 32Mbit Flash RAM, and the experiment FPGA. It is contained in a Ceramic

Quad Flat-Pack (CQFP) 228 pin package. Relevant FPGA specifications are listed in Table 1. The control FPGA has several functions, including communication between the PC/104 computer and the X2 experiment FPGA and initializing and checking the X2 configuration.

Device	Equivalent System Gates	CLB Array	Logic Cells	Maximum Available I/O Pins	Block RAM Bits
XQVR600	661,111	48x72	15,552	162	98,304 (24 4Kbit devices)

Table 1. XQVR-600 FPGA Specifications [4]

2. Experiment FPGA

This FPGA is directly connected to the control FPGA and the six Elpida HM5225405B-75 256Mbit PC133 SDRAMs. It is also directly connected to the Intel 32Mbit Flash RAM. The experiment FPGA is intended to hold experiments for testing. The flight board also uses an XQVR-600 Virtex FPGA. As mentioned earlier, some versions of CFTP use a much larger Virtex II FPGA as the experiment FPGA.

3. XQR18V04 Configuration Flash PROM

This PROM contains the X1 Control FPGA configuration information. X1 automatically loads the configuration at power-on as per normal Xilinx FPGA procedures. Most of the available 4Mbit space is used to hold X1 configuration data. Future projects might use the extra memory for program storage by adding the configuration PROM as another X1 device.

4. Intel 28F320C3 32Mbit Flash RAM

This RAM contains configuration information for the X2 experiment FPGA. The XQVR-600 FPGA requires 3,607,968 bits for a full configuration; there is enough

storage in this device to hold nine different X2 configurations [5]. X2 is configured via a SelectMap configuration process implemented in a functional block in X1 and not via an automatic Xilinx startup process as in X1.

5. 6 Elpida 256Mbit PC133 SDRAMs

This SDRAM is connected to the experiment FPGA (X2) and is not directly accessible by X1. It has a total capacity of 192 megabytes; less capacity would be available for use in a fault tolerant configuration. The version of CFTP with a Virtex II experiment FPGA does not have SDRAM.

C. SUMMARY

This design was documented in theses by Ebert and Johnson [6], [7]. Flight experiments are described in theses by Coudeyras and Caldwell [8], [9]. Chapter III describes the functions that the current version of the X1 configuration performs and how they will be modified and enhanced with a microcontroller architecture.

III. MICROCONTROLLER DESIGN

A. OBJECTIVE

The CFTP microcontroller should implement fault tolerance in the X1 control FPGA as well as adding additional functionality associated with programmed code and software-controlled modules. While the requirement for fault tolerance is clear, the choice to change to a microcontroller requires more justification.

Using programmed code gives many advantages to X1 over fixed-function hardware. Firstly, it allows module reuse. An interface to a device may be used for more than one process without hardware duplication. Secondly, with a centralized control structure adding additional modules only requires interfacing with the microcontroller. Busses and control lines between functional blocks are reduced or eliminated. Implementing complicated functions is made easier as programmed code requires less effort to develop compared with designing large state machines. Finally, using programmed code makes it easier to implement changes to X1's operation since programming in C is easier than programming in a Hardware Description Language (HDL). CFTP experimenters don't have to understand how all the X1 hardware works−only how the hardware interfaces operate with the microcontroller. The main disadvantage of using a microcontroller is that it will likely take more clock cycles to complete operations than its fixed-function HDL equivalent.

Besides simply passing data between the PC/104 computer and X2, X1 can now function as a full-fledged data processing unit and perform tasks such as formatting status reports or implementing a full-featured communications protocol. It should even be possible to eliminate the PC/104 computer board and have the X1 microcontroller communicate directly with the CDH or with the satellite downlink. Figure 2 contains a block diagram of the main functional blocks of the experiment board while Figure 3 shows the configuration after the microcontroller is implemented.

B. FUNCTIONAL BLOCKS

There are five main functional blocks in the fixed-function X1 configuration. They are the PC/104 interface, 64-bit timestamp counter, SelectMap configuration module, SelectMap readback module, and X2 interface. These blocks were implemented as Very-high-speed integrated circuits Hardware Description Language (VHDL) modules, with the configuration data for the FPGAs generated by the Xilinx ISE Computer-Aided Design (CAD) tool [10]. Figure 2 shows the block diagram of X1 before the microcontroller was added.

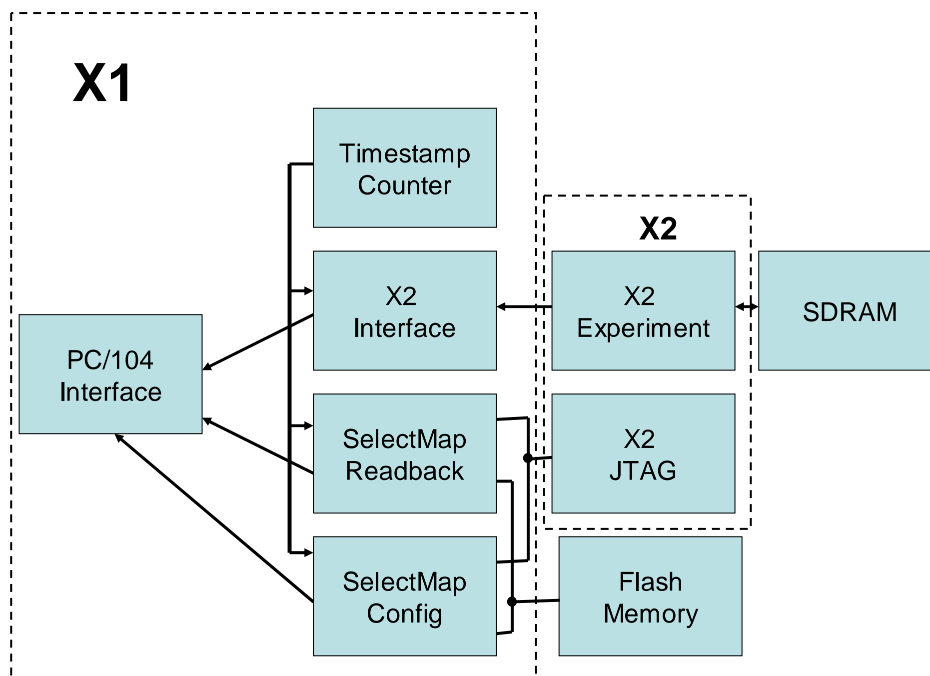

Figure 2. CFTP Block Diagram – Before Microcontroller

Figure 3 shows the block diagram after the microcontroller was added. Some functional blocks were rewritten in Verilog while others were only slightly modified and left as VHDL modules. Although it would have been possible to eliminate the SelectMap modules and perform their tasks in software by adding the 32 Mbit Flash memory and X2 SelectMap port as microcontroller I/O devices, they were left as separate hardware processes. This conversion was not made for several reasons. Firstly, the modules were

already designed and tested. Secondly, implementing the SelectMap readback module in programmed code could have used considerable processor time, possibly interfering with the main task of the X1 processor; communication between the PC/104 computer and X2. A brief description of the hardware modules, the tasks they perform, and how they were modified for use with the microcontroller follow.

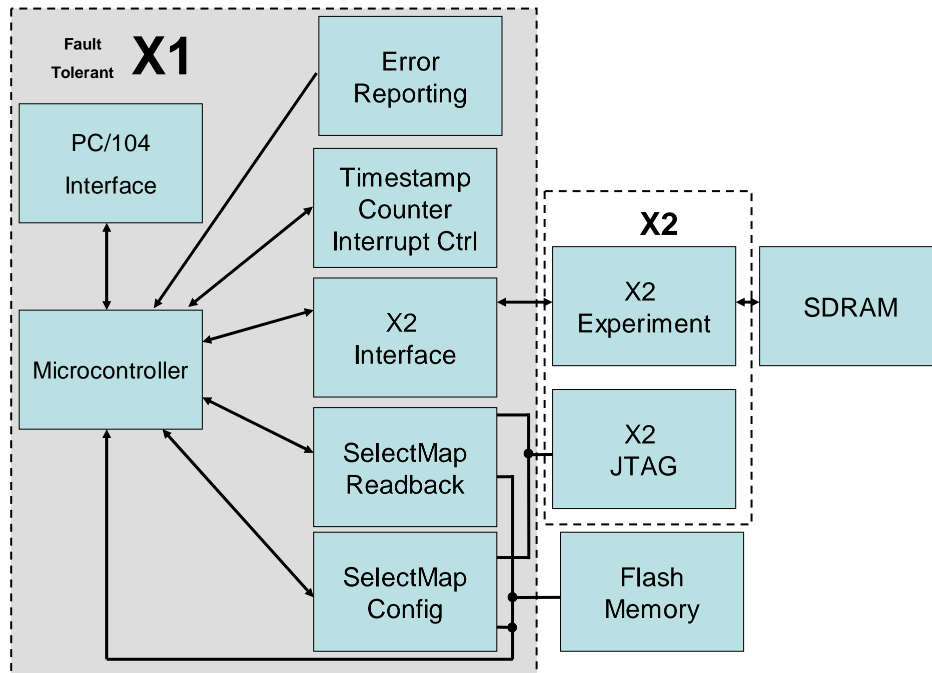

Figure 3. CFTP Block Diagram – After Microcontroller

1. PC/104 Interface Module

The PC/104 interface module allows bidirectional 8-bit data transfer between the PC/104 computer and X1. The PC/104 computer is in control of the PC/104 bus, using different addresses for enabling and disabling interrupts, performing data transfer, and checking the status register used for handshaking. The X2 interface module and the two SelectMap configuration modules connected directly to the PC/104 interface module.

This module was modified for use with the microcontroller by severing all the direct connections to the other modules. All PC/104 bus communication is coordinated by the microcontroller. The source code for the PC/104 module is listed in Appendix A, Sections W (xsoc_pc104.v) and K (pc104queue.v).

2. Timestamp Counter

The 64-bit counter in the Timestamp Counter module was used in the original version of X1 to generate timestamps for inclusion in status messages sent from CFTP to the PC/104 computer. Status message time was calculated using the 64-bit counter value included in the message and the CFTP startup time recorded in the PC/104 computer. The two SelectMap modules and the X2 interface were directly connected to the timer output since all these modules generated status reports.

This module was modified for use with the microcontroller by severing the direct connections to the other modules; it is now connected only to the microcontroller. To allow a choice of a smaller counter to save FPGA resources, the source code was changed to implement a 32, 48, or 64-bit counter selectable via `define statements. Additional functionality was added to the counter by using output bits 16-31 (selectable through software) to periodically generate interrupts, forming a basic timer. An interrupt controller was also inserted in the module to process the interrupts from the timer and other I/O devices connected to the microcontroller. The source code for this module is listed in Appendix A, Section U (xscounter.v).

3. SelectMap Configuration Module

This module initializes the X2 FPGA by downloading the configuration information into X2's configuration memory. The configuration information is stored in the 32Mbit Flash RAM.

This module was only slightly modified from the original; the status message it output to the PC/104 interface was replaced with a status byte the microcontroller could use to determine when configuration was finished and a flash memory base address was

added so multiple X2 configurations could be selected. The source code for this module is listed in Appendix A, Section O (selectmap_config_xsoc.v), and Appendix B, Section A (selectmap_config.vhd).

4. SelectMap Readback Module

This process is done to detect errors caused in X2's configuration by SEUs. This module reads the configuration from the X2 experiment FPGA and compares it with known good values in the 32Mbit Flash RAM. If any bits are found to be in error, a status report is sent via the PC/104 interface.

This module was only slightly modified from the original. Direct status messages to the PC/104 bus were eliminated; instead the data values they contained were made available to the microcontroller via memory-mapped registers. The source code for this module is listed in Appendix A, Section P (selectmap_rb_xsoc.v), and Appendix B, Section B (selectmap_readback.vhd).

5. X2 Interface

The X2 interface was designed to be modified for each individual experiment. It includes all the experiment-specific portions of X1. Experiments to date had all used data produced in X2; results were sent to the PC/104 bus via an interface to the PC/104 interface module.

A sample X2 interface was specified but was not implemented for the microcontroller since creating an experiment was not the objective of this thesis. However, the microcontroller can easily move data in and out of an experiment, including using data from the PC/104 bus or even generating the input data in the microcontroller itself. Data coming from the experiment can be formatted, buffered, and sent to the PC/104 computer as desired.

C. SUMMARY

More details on the design and characteristics of the new modules are contained in Chapters VIII, IX, X, XII and XIII, and source code is contained in Appendices A and B. After it was established that a microcontroller architecture would improve CFTP, the next step was establishing the selection criteria for and choosing a processor to use. That process is described in Chapter IV, Processor Selection and Features.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. PROCESSOR SELECTION AND FEATURES

A. SELECTING A PROCESSOR

Selecting an existing soft-core processor eliminated the time needed to test and implement a new design. Desirable features included a robust instruction set, comprehensive documentation, operating system, and C compiler. Most importantly, the processor had to fit in the control FPGA. 32-bit designs were quickly eliminated since none were found that were small enough. Many 8-bit designs were rejected since they were designed to emulate an existing CISC processor or microcontroller and were also too large. Only one design was found that had the necessary functionality and was small enough. That processor was the XSOC processor developed by Jan Gray as documented in his "Building a RISC CPU and System-on-a-Chip in an FPGA" series of articles that was published in Circuit Cellar magazine [3]. The XSOC License Agreement is included as Appendix E; as this thesis and provided source code rely extensively on XSOC it is also subject to the agreement. Specific applications of the CFTP microcontroller should carefully consider the XSOC License Agreement to ensure compliance.

B. XSOC GENERAL DESCRIPTION

As developed by Gray, XSOC is not just a soft-core processor. It is a design for a complete "System-on-a-chip" that includes source code for a 16-bit processor, video controller, combined memory and I/O controller, and sample I/O devices. It was originally developed for the XESS XS40-005XL series of FPGA development boards. These boards included a 32KB asynchronous SRAM, VGA port, 7-segment display, and a parallel port. Programs were loaded directly into the SRAM via a connection to an attached personal computer. XSOC came with excellent documentation, a modified version of the lcc retargetable C-compiler, an assembler, and a simulation tool.

None of the I/O modules included with XSOC were directly applicable for use in X1. The parallel port proved useful during early development and testing. The memory controller was modified extensively for use with the block RAMs present in the X1

FPGA. Finally, XSOC was modified to include fault tolerance; TMR was used for all systems except main memory which was implemented using ECC.

Two complete source options were provided with XSOC; Verilog HDL text files and Xilinx ISE Student Edition schematic files. Unfortunately, the Xilinx files were created with a now-obsolete edition of ISE and currently supported versions of ISE will not read them. The Verilog files are plain-text and are readily compiled using the Xilinx design tools.

The XSOC processor is a 16-bit RISC design. It features a 3-stage pipeline, a 16-entry register file, and 43 distinct instructions. Figure 4 shows the processor data path. There is no status register; instructions that rely on condition codes must immediately follow those that set them. A complete instruction set summary is given in "The xr16 Specfications" manual [11]. The instruction set and basic processor architecture was left unchanged for use with the CFTP.

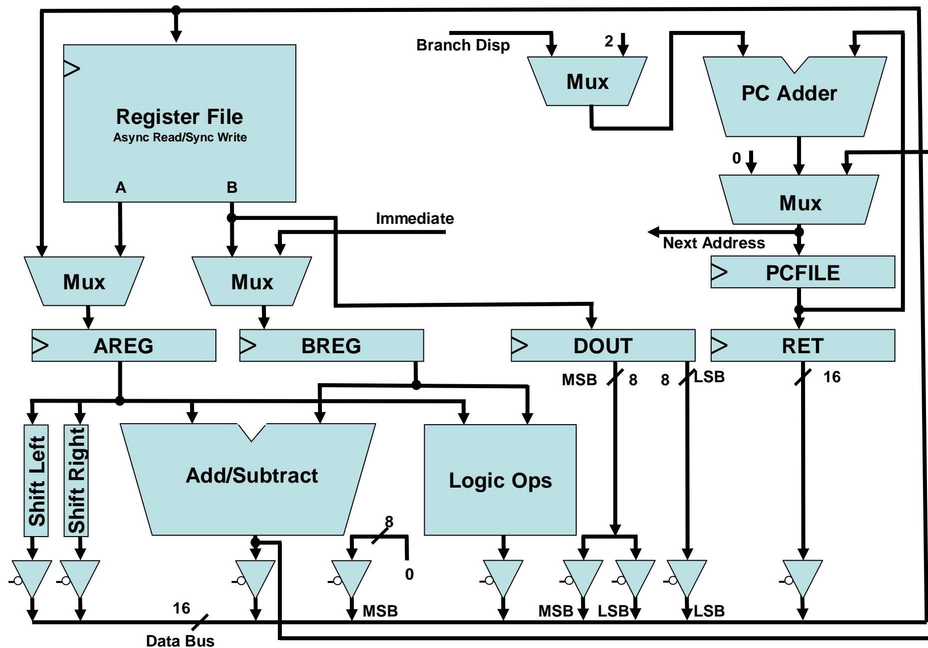

Figure 4. XSOC Data Path

Figure 5 shows the pipeline execution. There are three stages in the pipeline: instruction fetch, instruction decode, and execute/writeback. In the instruction fetch stage, the next instruction address is presented to memory. This operation is then followed by the instruction decode stage where the results from the previously requested memory access are made available to the instruction decode circuitry in preparation for the execute/write back stage. At this point the A and B operands are retrieved from the register file, immediate data, or data from the previous writeback/decode stage. Finally, the results are computed and written back in the execute/write back stage. Memory accesses require at least one additional clock cycle and will cause a pipeline stall. Taken branches and jumps will also cause pipeline stalls.

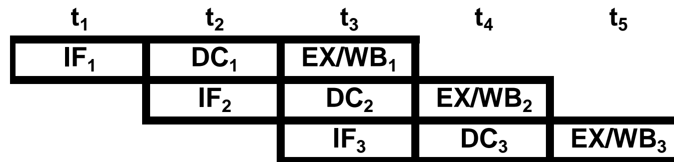

Figure 5. Pipelined Execution. From [3]

C. SUMMARY

After selecting XSOC as the basis for this project, the major effort of this thesis was modifying it for fault tolerance. This process is described in Chapter V.

THIS PAGE INTENTIONALLY LEFT BLANK

V. IMPLEMENTING FAULT TOLERANCE

Fault tolerance was implemented through both Error Correction Coding (ECC) and Triple Modular Redundancy (TMR). ECC was used in designing the memory subsystem and is discussed further in Chapter VIII. This chapter presents how TMR was implemented in X1, methods to implement TMR using Xilinx ISE, and discusses important fault tolerance issues with the Virtex FPGAs used in CFTP.

A. TRIPLE MODULAR REDUNDANCY

The strategy for making XSOC fault tolerant through TMR was one of placing voters before each clocked storage element and triplicating the functionality of all the modules. If an error occurs within the combinational logic of one of the three copies, the error is not propagated to that copy's pipeline registers because it is rejected as the minority result. Since each register has its own input voter, a voter error will only affect that particular copy of the circuitry and will be corrected by the voter on the next pipeline stage. This organization is shown in Figure 6. Figure 6 is intended as a cutaway view of one of the pipeline stages; feedback paths are not shown.

The clock is considered a trusted source; an error in the clock may cause unintended operation. Ideally one would have three redundant clock signals but the existing CFTP hardware does not have that capability.

Bus and signal widths were increased by three times to accommodate the three copies of each module. Voters were instantiated as <signalname>_v, while voted signals were named v_<signalname>. This convention aided in the design process as it was obvious which signals were the result of a voted output.

Although the voters could have been placed before or after the pipeline registers, the deciding factor was that the synchronous block RAMs present in the FPGA have a clocked input. Placing the voters after the pipeline registers would have resulted in the signal propagating through two sets of voters on the pipeline stage terminating in the memory inputs; once after the pipeline registers and again just before entering the

memory inputs. Having more than one voter between pipeline stages is undesirable as it could lead to a decrease in the maximum operating speed of the microcontroller due to the delay of two voters rather than one.

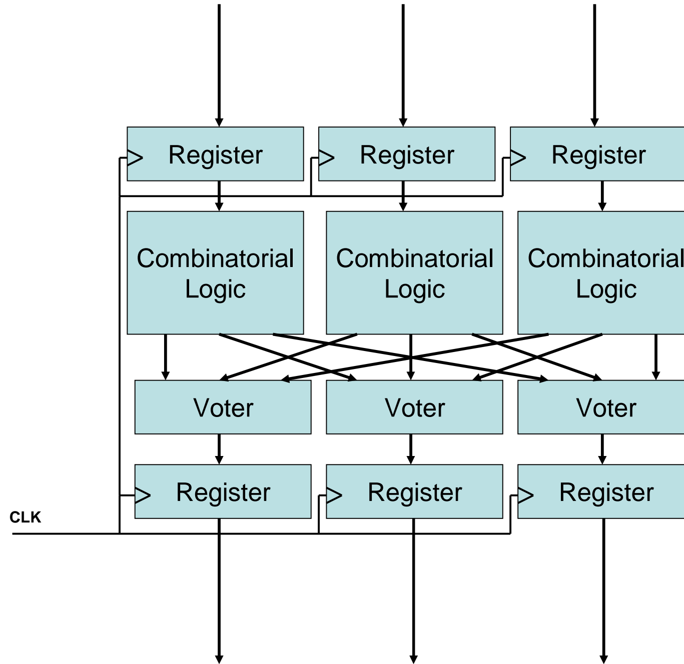

Figure 6. Voter Strategy

The voters used are bitwise majority voters. A block diagram of one bit of a voter is shown in Figure 7. Voters are instantiated through use of the *voter1nsyn* module. This module accepts the number of result bits n as a parameter. Data in is $3n$ bits wide, data out will be n bits wide. Data on the input bus is organized with the least significant third of the bits designated as input A, the next third designated as input B, and the most significant third designated as input C. Error indication is provided through the global error syndrome bus as described in Chapter XII. The error indication is limited to errors on a particular input A, B, or C and will not indicate which particular bit is in error.

Each bit on the input bus is treated individually, with one bit from each of the three A, B, and C inputs sent into the structure shown in Figure 7. An error indication for input A, B, or C is asserted if that particular input's bit does not agree with the other two inputs. The A, B, and C error outputs from each bit are ORed with the A, B, and C error

outputs of the other bits in the voter into three A, B, and C error indication bits. These bits are output to the global error syndrome bus on the next clock cycle.

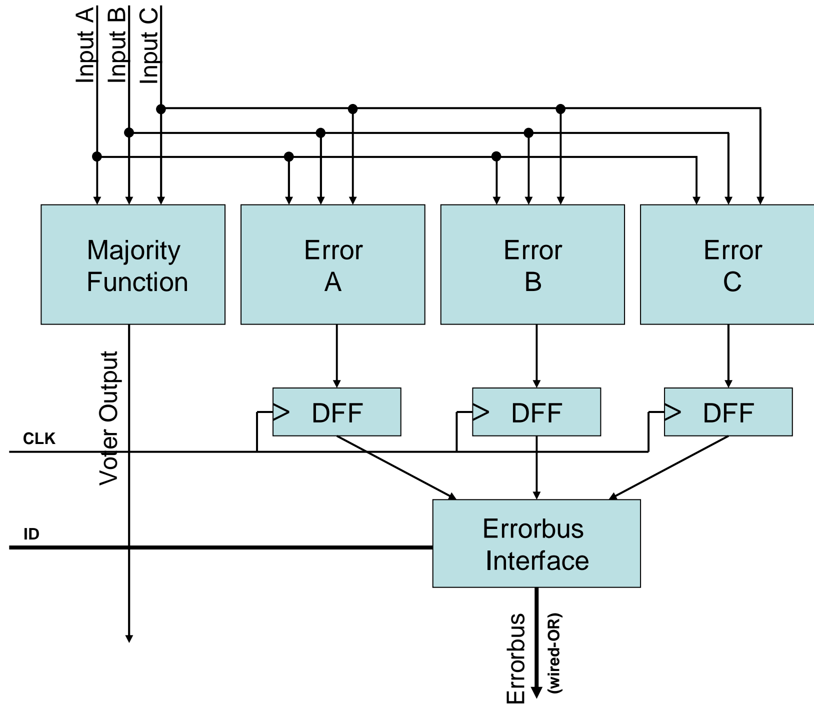

Figure 7. Voter Operation

Two different versions of the error detection voter were developed. The initial version ORed all the like A, B, and C error bits together and then saved the results in three flip-flops at the positive edge of the clock. This design was later improved by moving the OR circuitry after the flip-flops which resulted in a faster maximum system clock due to a shorter critical path. Unfortunately, this modification also used many more flip-flops and FPGA resources. When the design was nearly complete it was necessary to switch back to the slower voters due to running out of FPGA slices. Fortunately, this change did not result in a slower system clock since both choices were too slow for operation at 25 MHz and were fast enough for 20 MHz. (The clock is derived from the PC/104 bus SYSCLK which runs at 51 MHz. Dividing this clock by 2 results in a ~25Mhz clock; dividing by 2.5 results in a ~20Mhz clock)

Similarly, it was necessary to remove the error detection (but not correction) capabilities on the I/O devices attached to XSOC due to a lack of FPGA slices. They use a version of the voter (*voter1n*) that has the majority function as its only output.

B. XILINX ISE PARTITIONS

The CFTP microcontroller was created using Xilinx ISE 9.2i. The ISE software normally removes redundant logic during global optimization. Unfortunately redundant logic includes removing the redundant elements in TMR designs. In order to halt this behavior, one option is to use partitions. Although not the intended purpose, placing the redundant modules in individual partitions will stop the global optimization process from removing them.

Partitions are intended as a method to separate blocks of code that have not changed during the design process, resulting in shorter compilation times since only the changed partitions are recompiled. Alternatively, they may be used as a method to keep a "production-quality" module that has been approved for use from alteration after the quality assurance process [12]. When a module is implemented as a partition, optimization is done on the partitioned module which is then saved as a "black box" with only inputs and outputs visible to the global optimization process. This "black box" abstraction works well for use with TMR since the optimizer doesn't know the elements inside the boxes are redundant. Testing revealed that the TMR circuits were more than three times the size of the non-partitioned circuits as one would expect (the voters would account for the more than threefold increase). Additionally, the messages during compilation that redundant elements were removed were not observed after implementing the design using partitions.

Using partitions has some limitations, however. Tri-state busses are not allowed on connections external to the partition. (They are allowed internally, however). Tri-state outputs can be replaced by separate input and output ports and output enable signals on the instantiating module. Partitioned modules may not be instantiated by generate/for loops. This limitation makes the process of creating TMR on combined busses more prone to error as each bus connection must be specified manually. Lastly, partitions

should not use include statements [13]. Using include statements with ISE 9.2i does work with the expectation that changing the included code might not trigger an automatic recompile. Include statements are used to hold global constants in partitioned elements of this design with the knowledge that changing them should be followed by a complete recompile.

To create a partition in ISE, right-click on the module name in the Sources window (make sure the Synthesis/Implementation option is selected) and select *New Partition*. They may be deleted by selecting *Delete Partition*. There are several ways to inadvertently remove partitions. If the top-level module is changed or a module that contains partition data is removed from the project, partition data is removed and must be recreated.

C. COMPLETE FAULT TOLERANCE

Simply using TMR and ECC in a Virtex FPGA is not sufficient to guarantee fault tolerance. Configuration scrubbing, as used in X2, detects and corrects configuration errors. Configuration errors may change hardware functionality until corrected. Currently no configuration scrubbing is performed on X1. Additionally, some errors are persistent; they are not corrected with a configuration scrub. Specifically, "keeper" circuits may be created on inputs to FPGA elements that are permanently set to high or low values. The "keeper" circuits are created with otherwise unused logic that is not included in configuration scrubbing [13]. These errors require a power-on-reset to correct. Xilinx XAPP187 discusses these issues further. Both problems are candidates for future work. Finally, X1 on current CFTP hardware will never be completely fault tolerant since there is only a single system clock and most I/O connections are neither redundant nor error corrected.

D. SUMMARY

Implementing TMR and ECC is an important first step in achieving true fault tolerance in X1. Removing "keeper" circuits and adding configuration scrubbing on X1 are the final two steps required to make X1 as fault tolerant as possible and should be addressed by future work. Using partitions allows TMR implementation with the Xilinx ISE tools. Chapter VI describes the modifications that were made to the XSOC SoC for TMR conversion.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. MODIFYING XSOC FOR CFTP

The basic concept of XSOC was unchanged. Functions implemented using distributed RAM were replaced with an equivalent function created with flip-flops, the memory controller was modified for use with the on-FPGA block RAMs, and the entire design was made fault tolerant through the use of TMR. This chapter describes the first two changes while memory controller modifications are described in Chapter VII.

Xilinx XAPP216 notes that when using configuration scrubbing on Virtex FPGAs, elements that are implemented with distributed RAM cannot be used [14]. This limitation occurs because distributed RAM uses the configuration memory as RAM; the scrubbing will "correct" the RAM contents back to power-on values. The original XSOC register file and program counter were created using distributed RAM. Even though configuration scrubbing is not currently active on X1 the intent was to prepare the design so it could be implemented later. Additionally, the design was tested as an experiment on X2 during early development which would not have been possible had the original distributed RAM parts been left in place.

A replacement register file was created using groups of D-type positive-edge clocked flip-flops and three decoders. Figure 8 contains a block diagram of the new register file. The flip-flops are arranged in groups of 16, with all groups sharing a common input and the outputs connected to two tri-state output busses. Two decoders were used to select which group would drive the A and B output busses. The third decoder was used to determine which group received a clock enable for store operations. This design operates with an asynchronous read and a positive edge-clocked write-back. Since register 0 is defined in the architecture as a constant zero value, group zero is not connected to a register and instead connects to tri-state drivers sourced to ground.

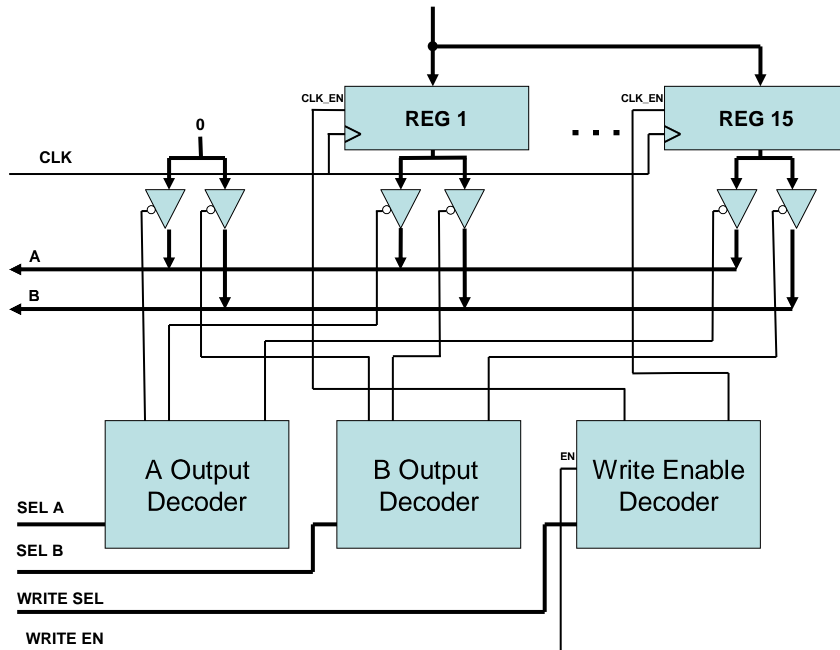

Figure 8. Replacement Register File

The original XSOC program counter was also implemented with the same structure as the register file but used only two of the 16 available registers. The first entry was the program counter for normal operation while the second was the DMA program counter. This design was replaced with two 16-bit registers and a multiplexer. There is no DMA support with the current memory design but the DMA program counter was implemented in case DMA is re-implemented in the future.

Implementing TMR, while a simple concept, proved more difficult in practice than initially expected. Every signal had to be expanded. Every register internal to modules added two signals to the module's port list and added a voter to the module. Instantiating some modules was very tedious due to the large number of ports and varying bus indices on each of the three copies.

Chapter VII describes the new memory design and changes that were made to the XSOC memory controller.

VII. MEMORY SUBSYSTEM

A. DESIGN

The available memory for the microcontroller was limited to the amount of block ram on the Xilinx XQVR600 FPGA, which is 24 4Kbit dual-port synchronous block RAMs. The original XSOC design used an asynchronous 32Kbyte 8-bit static RAM that was included on the XESS development board. Design goals were to make the system as fault tolerant as possible while maximizing use of the on-chip RAM and the operating speed of the system.

With only 24 4K-bit RAMs on board, fault tolerant design choices were limited. The RAMs are dual-ported, allowing operation as 48 2Kbit RAMs. Two fault tolerant options were explored: a TMR 2K-word system and a Hamming-coded 4K-word system.

The 2K-word system offers the advantage of speed and simplicity at the expense of storage capacity. The 4K-word system offers higher capacity at a slower speed with a greater complexity. The 4K-word (8K-byte) system was chosen to maximize the amount of RAM available to the XSOC programmer.

B. XSOC MEMORY MODIFICATIONS

The original XSOC design used the DOUT register driven to the data bus for processor memory writes, and used the output of the XA register for the address for reads and writes. The existing datapath was altered for use with block RAM. Figure 9 shows the new datapath outputs to memory. The synchronous memory required valid data a clock cycle earlier than the previously used asynchronous RAM in order to eliminate additional wait states. This constraint caused the need for a separate bus from the processor to memory for data writes since the processor data bus was possibly in use on the previous cycle to the write operation. Fortuitously, the block RAMs also had separate data inputs and outputs which eliminated contention from adjacent read cycles to memory.

Both the DOUT and XA registers are controlled by the MEM_CE clock-enable signal from the control path. This signal is deasserted during memory pipeline stalls. Multiplexers were added to select either the input values (no stall) or output (stall) values of the DOUT and XA registers to obtain the data one clock cycle earlier. The operation of the DOUT register and its associated drivers was unchanged.

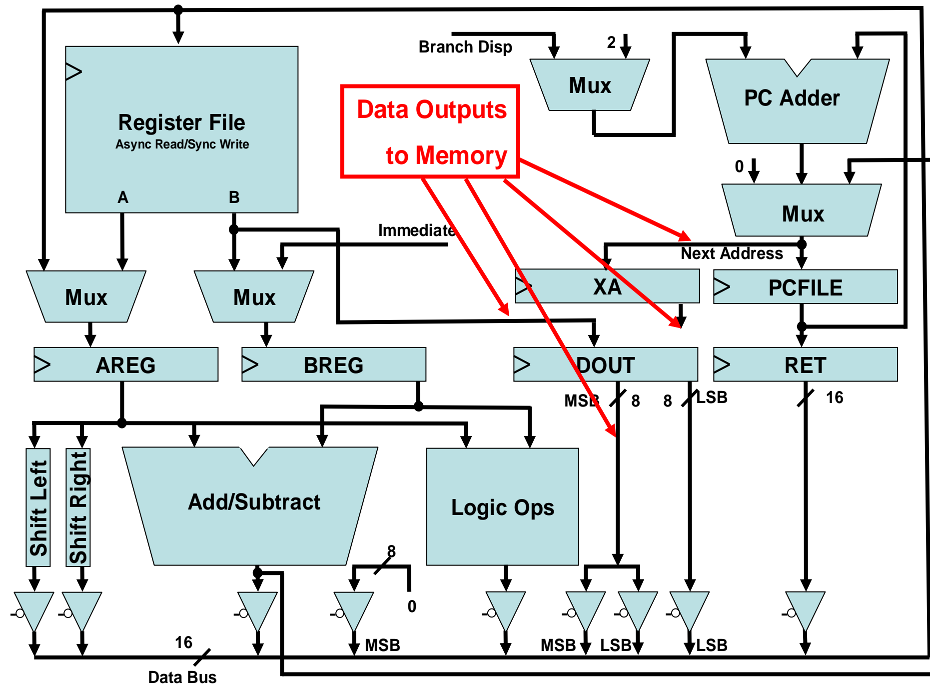

Figure 9. Datapath Outputs to Memory

The memory controller source code (memctrl.v) was modified for the new design, as well as the datapath (datapath.v) and main xsoc.v file.

C. MEMORY READ CYCLE

The memory read cycle has two modes of operation; read word and read byte. Both operations start in the instruction decode stage with the next memory address *NEXT_ADDR* and write enable *WE* deasserted. The output of the memory is then available on the next clock cycle.

All reads are done as a word with the results from the 22 BRAMs input into the ECC decoder where error correction is applied if necessary and presented to the two output busses. There are two output busses from main memory; one is for the pipelined instruction fetch; it is a 16-bit direct input to the control module. The other is the tri-state output to the data bus.

Data is addressed in bytes; the least significant bit is not presented to the memory modules. Instead, it is used to select the high or low byte after fetching the entire word. A multiplexer is inserted on the low byte to the data bus to select either the high or low byte. The high byte out of memory is always connected to the high byte of the data bus since all word reads must be word aligned. On byte reads, only the selected byte is driven to the low byte of the data bus. On word reads, both bytes are driven. Figure 10 shows the memory data flow.

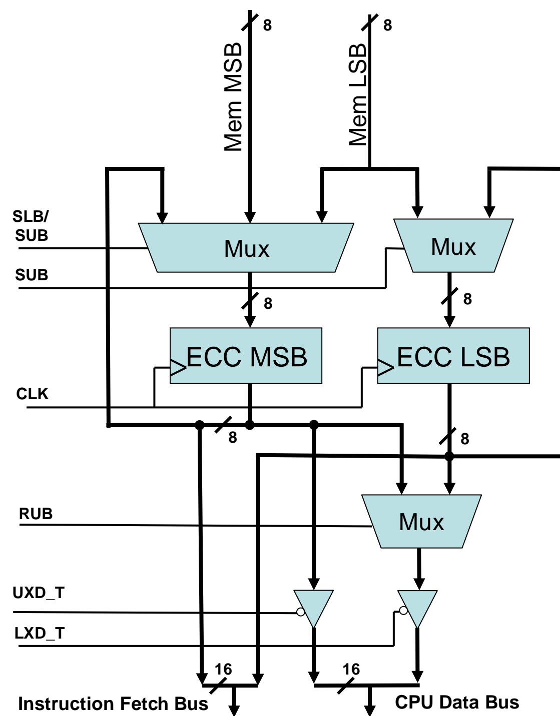

Figure 10. Memory Data Flow

D. MEMORY WRITE CYCLE

Memory writes are either byte or word writes, with byte writes taking two clock cycles and word writes taking one. Single-cycle byte writes would have been desirable but would require more block RAMs than are present on the FPGA. More RAMs would have been required due to the additional parity bits generated when memory is implemented as two 8-bit ECC bytes rather than a single 16-bit ECC word. Memory writes are always completed as a word write. This operation is necessary since writing the high or low byte by itself would not properly update the six parity bits (the parity is for the entire word, not just one byte). Additionally, since the value of the byte not being written is unknown, a write byte operation requires a read cycle beforehand to obtain the value of the byte not being written to properly calculate the parity for the word.

For a word write, the data is input from the B output of the register file to the ECC encoder. The encoded values are then sent through a voter to the data inputs of the BRAMs, which capture the data on the next clock.

For a byte write, the data is stored in the MDR register during the pipeline stall caused by the additional wait state. The value of the next address *ADDR_NXT* is read and sent through the ECC decoder for error correction. It is also latched into a holding register to preserve the address for the next cycle. On the next clock cycle, the value of the byte that was not to be written is sent to the ECC encoder along with the new byte of data. This newly formed word and associated parity bits are then written on the following clock cycle.

E. ERROR CORRECTION CODING

The memory subsystem uses Hamming codes to generate parity bits for error correction and detection. Table 2 shows the 5 check groups and their associated data bits. For a 16-bit word, six parity bits are needed for single error correction and double error detection. Memory is arranged with the order of the 16 data bits preserved in the first 16 bits of memory and the additional parity bits in positions 17 through 22. There are five Hamming bits with an additional sixth bit as an overall parity check. The parity

calculations are done by XORing the data bits selected. The grouping is as follows, with bit 0 being the least significant data bit and 15 the most significant:

Check Group	**Data Bits**
1	0,1,3,4,6,8,10,11,13,15
2	0,2,3,5,6,9,10,12,13
4	1,2,3,7,8,9,10,14,15
8	4,5,6,7,8,9,10
16	11,12,13,14,15
Overall	Data bits 0-15 and the 5 check group bits

Table 2. Parity Group Bits

Arranging the data bits with the check groups in the bit positions indicated would have made decoding easier; however the order of the 16 data bits was preserved to aid in the debugging process as well as to facilitate program reuse between ECC and non-ECC versions of the microcontroller. The six most significant bits of data are simply ignored for non-ECC versions.

Error detection and correction is accomplished by XORing the corresponding check group bit with the data bits that generated it. The check bits are then combined into a syndrome that is sent to a modified decoder. The decoder outputs a 16-bit value that is all zeros when no errors are detected. If an error is detected the syndrome input to the decoder will cause the the bit that corresponds to the erroneous memory bit to be asserted. The output from the decoder is then XORed with the output of the low 16 bits of memory resulting in correct output to the instruction and data bus, if applicable.

The design of the memory system prevents SEUs from causing memory errors. Figure 11 shows the memory organization. Traditional ECC designs are protected against data errors but not addressing errors. To solve this issue, individual voters are placed at all inputs of the 22 block RAMs. The only common input is the clock. Any

single bit failure will affect at most one RAM, including a failure on one of the address lines or the write enable. The ECC decoders will correct the erroneous bit and the correct value will be presented to the three data buses.

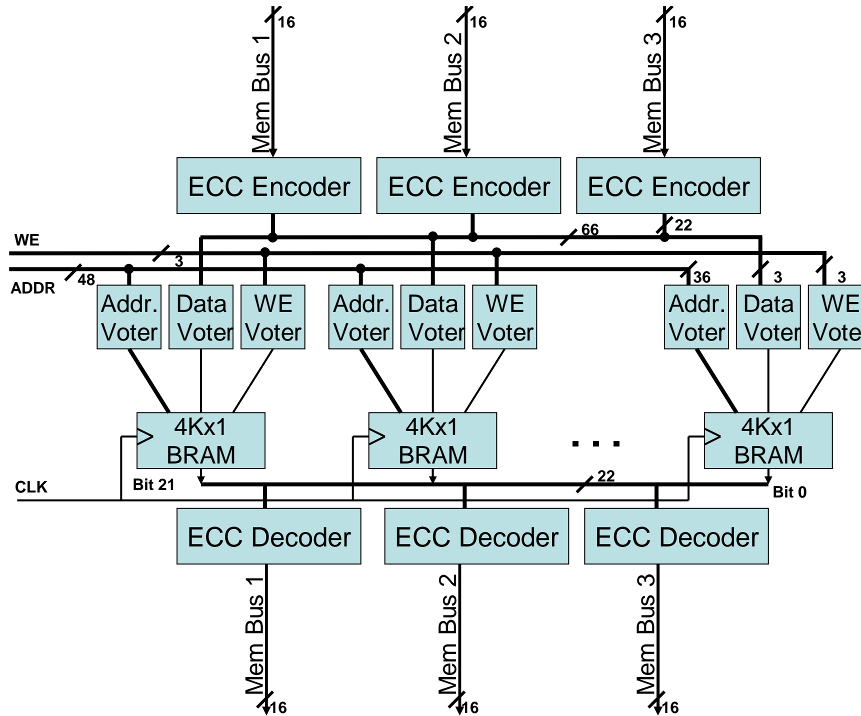

Figure 11. Fault Tolerant Memory

F. ALTERNATIVE MEMORY OPTION

The alternative memory option mentioned in the beginning of this chapter was to use both ports of the dual-ported 4K block RAMs, dividing them into 48 2K RAMs by setting the high address bit on one port to high and the high address bit on the other port to low. Figure 12 shows the memory data flow for this design. This option eliminates the extra wait state on write byte caused by reading back the entire word as well as the loop-around circuitry associated with it. It also requires fewer voters since each block is triplicated; redundancy is by instance and not bitwise. Figure 13 shows how the memory structure would have been organized. Only three address (12-bit), six write-enable (one-bit), and three data (16-bit) voters are required. It eliminates the possibility for a background memory scrub since both ports of the memory are in use. Note that there are

write-enables for each byte, as opposed to the single write-enable for the word on the ECC system. Also of note is that this design does not offer more protection against SEUs than does the ECC version for CFTP use. This equality does not hold when using a platform with redundant clocks; the single clock is the weakness of the ECC version.

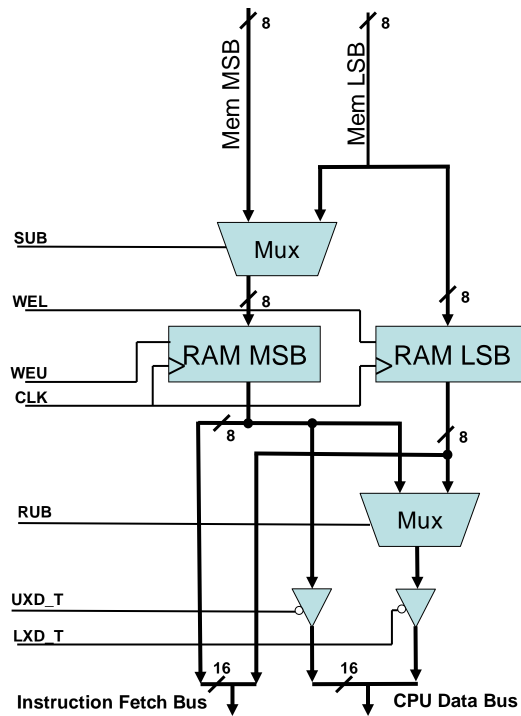

Figure 12. Alternative Memory Data Flow

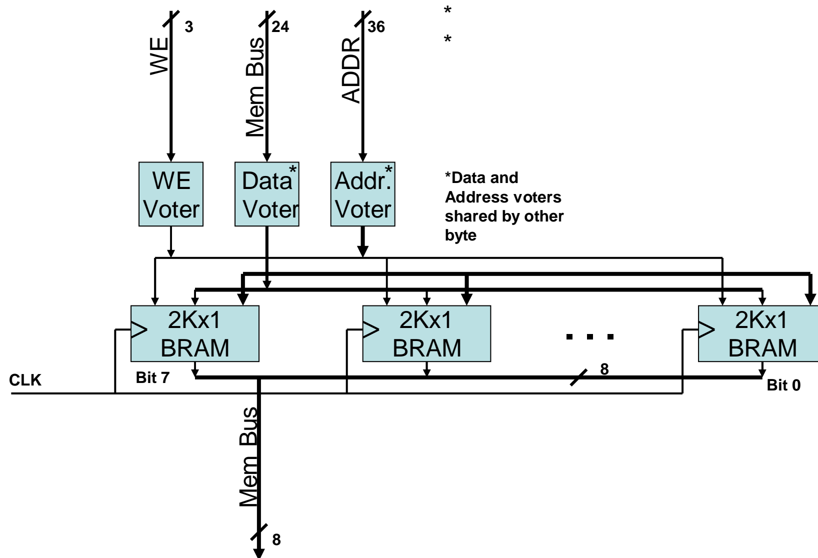

Figure 13. Alternative Design for Fault Tolerant Memory

In this chapter, two memory subsystem alternatives were presented. Each has advantages over the other. With the available resources in the Virtex FPGA the ECC version was selected. Were this project ported to a newer FPGA with more or larger block RAMs, the TMR solution would have been much more attractive due to the simplicity of the design, single clock cycle byte writes, and faster operation.

G. MEMORY SUBSYSTEM FILES

The memory subsystem is instantiated in 3 Verilog files: bram_ecc.v, memio.v, and bram4kx22.v. Bram_ecc is called from the top-level xsoc.v file and contains the error-corrected memory subsystem module, the three memory I/O modules, and the block RAMs.

The revised memory controller and ECC RAM design were the last revisions made to XSOC and completed the basic microcontroller design. The next chapters describe microcontroller I/O, including both adding new modules and revisions to the previous X1 functional blocks. Chapter VIII starts the process by discussing adding I/O modules.

VIII. ADDING I/O MODULES

This chapter is intended to serve as a guide for adding new I/O modules. The first three sections deal with the three sections of a TMR module: instantiation of the complete module, the wrapper module, and the triplicated functional block of the module. The fourth section describes an I/O module interface for an experiment located in the X2 experiment FPGA. Source code for the sample I/O module is found in Appendix A, Section H (iotemplate.v).

A. TOP-LEVEL INTERFACE

Adding I/O modules to the X1 design is accomplished by adding the module to the top level x1control.v file. The module will consist of a "wrapper" function that instantiates the XSOC *ctrl_dec* and the actual module code. A sample I/O module is included in Section H of Appendix A and will be used to discuss implementing an I/O module. Table 3 contains the typical ports that the top-level I/O module would use. The X3 column indicates that the signal is a TMR bus and is actually three times the stated bus width.

A brief discussion of the module signals is in order. *CLK* and *RST* are the global system clock and reset signals, respectively. *CTRL* is used by the XSOC *ctrl_dec* module to create other control signals used in the module. *CTRL0* is driven low if extra wait states are needed before the module completes an operation. It will halt the pipeline until it is driven high. If using this signal, comment out *assign ctrl0 = {1'b1, 1'b1, 1'b1};* in the top level-module. Additionally, realize that this is a single input pin and extra logic will need to be inserted if more than one module needs to use it. This pin is part of the original XSOC CTRL specification as bit 0; mixed bus types are not allowed so it was implemented separately.

SEL is used to indicate the memory region that activates the IO module. In the original design, this was a single select pin chosen from an 8-bit bus, supporting eight I/O modules. Bus indices 0 to 7 correspond to memory regions starting at FF00 with each ID

being 20h of memory space higher. This convention is still used, but since the bus is triplicated ID 0 is now pins 0, 8, and 16 of the bus. Send the three signals associated with the desired ID to the module as the *SEL* input.

INT_REQ is selected in the same way as *SEL* with 8 possible IDs on a triplicated bus. Note that unused *INT_REQ* pins are assigned to zero in the top level module and the assign statements must be uncommented if using those interrupt request lines.

D is the microcontroller tri-state data bus, while ERRORBUS is the global error reporting bus.

Signal Name	Input/ Output	Bus Width	X3	Description
CLK	Input	1	N	System clock, ~20Mhz
RST	Input	1	N	Global power-on reset
CTRL	Input	16	Y	XSOC IO control bus, from memctrl. Used by ctrl_dec_syn
CTRL0	Output	1	Y	Pin 0 of CTRL, drive low to indicate extra wait states
SEL	Input	1	Y	Driven high by memctrl when that memory region is active
INT_REQ	Output	1	Y	Drive high to indicate an interrupt request
D	In/Out	16	Y	Bidirectional processor data bus
ERRORBUS	In/Out	16	N	Global error syndrome reporting bus

Table 3. Typical IO Wrapper Module Ports

B. WRAPPER MODULE

The wrapper module instantiates three XSOC ctrl_dec control decoder modules, three "user" I/O modules, and the tri-state drivers necessary for outputting data to the data bus. Three copies of the modules are used to implement TMR.

The XSOC ctrl_dec_syn control decoder uses the inputs from *CTRL* and *SEL* to decode the 5-bit IO address and the load and store data strobes. It differs from the original XSOC specification as it has been modified for TMR operation. Both the CTRL and SEL busses are triplicated and voted upon in the decoder, with voter results reported on the global syndrome bus.

The tri-state output drivers are placed in this module since Xilinx partitions do not allow tri-state inputs/outputs on partition ports [12]. The *ld_t* (lower byte data output enable) and *ud_t* (upper byte data output enable) are output from the ctrl_dec_syn control decoders and enable the drivers. A voter was not placed on the driver enables in case one processor copy was in an inconsistent state with the other two to avoid the possibility of multiple driven signals on the same bus. This design should not impact the fault tolerance of the system; if one copy is in an inconsistent state it will most likely be unable to properly process data driven to the bus. Since all control signals are reloaded at each clock cycle, the inconsistent copy will return to normal operation on the next clock.

C. I/O MODULE

The sample I/O module presented in Appendix A, Section H has a 16-bit register accessible at the base address (FF00 if ID 0 is selected, FF20 for ID 1, … FFE0 for ID 8.) The typical load and store module will have ten ports. They are listed in Table 4.

In the sample module, a store to the register is enabled when *LD_CE* is asserted and the module address matches the address assigned to the register. Note that byte operations will assert only *LD_CE* and not *UD_CE*, so to be technically correct to the specifications the enable should be split between the high and low bytes. In this example, only word operation is desired so the *UD_CE* signal is ignored. The *LD_CE*, *ADDR*, and *D* signals are all voted upon since it is a clocked store operation.

For reads, the *LD_T* and possibly *UD_T* signals are asserted. These signals are not used by the simple module in the example since the tri-state drivers are in the wrapper module. They would be used in the wrapper module for cases where a read operation caused additional functionality besides a data transfer to occur, such as a read from a queue that causes the next item to become available.

Signal Name	Input/ Output	Bus Width	X3	Description
CLK	Input	1	N	System clock, ~20Mhz
RST	Input	1	N	Global power-on reset
ADDR	Input	5	Y	Contains the 5 bits of address dedicated to that IO module; base address + addr = actual address
DIN	Input	16	Y	Data in to module
DOUT	Output	16	N	Data out of the module
ERRORBUS	In/Out	16	N	Global error syndrome reporting bus
LD_T	In	1	Y	Used to indicate a read operation from the module (lower byte driver enable)
UD_T	In	1	Y	Used to indicate a read operation from the module (upper byte driver enable)
LD_CE	In	1	Y	Used to indicate a store operation to the module (lower byte store clock enable)
UD_CE	In	1	Y	Used to indicate a store operation to the module (upper byte store clock enable)

Table 4. Typical IO Module Ports

Reading is accomplished asynchronously with the outputs of the different storage elements being selected via a multiplexer based on the address input and output to the wrapper module via the *DOUT* port. Signals to the *DOUT* port should not be voted; any reads from the data bus are always voted upon before they are stored. In this simple example a zero value is returned if the address does not match the address assigned to the single register.

D. X2 I/O MODULES

There are 44 general-purpose I/O pins between the X1 control FPGA and the X2 experiment FPGA. This number is too few for a complete 16-bit TMR I/O module. It would take an additional 27 pins to implement the interface with 16-bit I/O and a 5-bit address space. However, it is enough for an 8-bit TMRed module with a 4-bit address space. For an 8-bit implementation, the assignment listed in Table 5 might be used.

Signal Name	Input/ Output	Bus Width	X3	Description
CLK	Input	1	N	System clock, ~20Mhz
RST	Input	1	N	Global power-on reset
ADDR	Input	4	Y	Contains the 4 bits of address dedicated to that IO module; base address + addr = actual address
D	In/Out	8	Y	Data in and out of module
LD_T	In	1	Y	Used to indicate a read operation from the module (lower byte driver enable)
LD_CE	In	1	Y	Used to indicate a store operation to the module (lower byte store clock enable)

Table 5. Possible X2 I/O Interface Module

This interface would use all 44 available pins. Instead of one wrapper module, two would be used; one would be contained in X1 and the other contained in X2. An adjustment to the wrapper module on X1 would be required; the tri-state drivers and the triplicated modules would be removed. The wrapper module on X1 would contain the *ctrl_dec* modules and assign the signals indicated in Table 5 to I/O pins. A second wrapper module for instantiation on the X2 experiment FPGA would be created. It would include the triplicated I/O modules as well as tri-state drivers used to allow the partitioned I/O modules to drive the data bus.

E. SUMMARY

This chapter is a guide for adding I/O modules to the CFTP XSOC-based microcontroller. Specifically, instantiating a new I/O module in the top-level x1control file and the contents of the wrapper and I/O modules were detailed. The next chapter describes the operation of the modified X2 Configuration module with the microcontroller.

IX. X2 CONFIGURATION MODULE

A. GENERAL DESCRIPTION

The X2 Configuration Module consists of two files; the original VHDL code written by Mindy Surratt and a wrapper function for the XSOC interface [15]. It reads data from the Intel 28F320C3 32Mbit Flash RAM and uses it to configure X2 via a dedicated connection. Source code for this module is located in Appendix A, Section O (selectmap_config_xsoc.v) and Appendix B, Section A (selectmap_config.vhd).

B. XSOC INTERFACE

The X2 configuration module has four memory-mapped ports. They are listed in Table 6.

Address (Hex)	**Function**
0x00	Configure Request
0x02	Flash Load Base Address (Low Word)
0x04	Flash Load Base Address (High Word)
0x06	Status

Table 6. X2 Configuration Module Address Map

To configure X2, the following steps must be followed. First, write the high and low words of the flash base address to their respective port addresses. If not specified the address defaults to zero. Next, write a one to the *Configure Request* address to initiate the configuration process. Finally, poll the *Status* byte until it returns a zero value.

C. VHDL CODE

The original VHDL selectmap_config.vhd module was left intact with two changes. The first modification was commenting out all the direct access to the PC/104 port. The status message that was displayed is easily recreated by polling the status byte

and using the processor to send the configuration complete message. The second modification was adding the flash memory base address option.

In summary, this chapter described the implementation and operation of the X2 Configuration Module. The next chapter describes the X2 Configuration Readback Module.

X. X2 CONFIGURATION READBACK MODULE

A. GENERAL DESCRIPTION

The X2 Configuration Readback Module consists of two files; the original VHDL code written by Mindy Surratt and a wrapper function for the XSOC interface [15]. It continuously reads configuration data from both the flash RAM and X2 and performs a comparison. If a disagreement is found an interrupt issued to the processor. Readback is resumed once the processor acknowledges the error. Source code for this module is located in Appendix A, Section P (selectmap_rb_xsoc.v) and Appendix B, Section B (selectmap_readback.vhd).

B. XSOC INTERFACE

The X2 Configuration Readback module has nine memory-mapped ports. They are listed in Table 7.

Address (Hex)	**Read/Write**	**Function**
0x00	Write	Readback Request
0x01	Read	Readback Status
0x02	Read	SelectMap Read Data
0x04	Read	Error Location High
0x06	Read	Error Location Low
0x08	Read	Error Word
0x0A	Read	Data Ready
0x0C	Write	Flash Load Base Address (Low Word)
0x0E	Write	Flash Load Base Address (High Word)

Table 7. X2 Configuration Readback Module Address Map

There are three addresses with write functionality. They are the two *Flash Load Base* addresses and the *Readback Request* address. To prepare to begin data readback, write the high and low flash load base addresses to their respective ports. The default flash load base address is 0x48. Note that this is NOT the same address as the flash load base address used by the X2 configuration module since the actual configuration data does not start at offset 0 from the FPGA initialization data generated by ISE. Writing a 0x1 to the *Readback Request* address will initiate the readback process.

If there is an error detected, reading the *Data Ready* status byte will result in a 0x1. Similarly the interrupt assigned to the module will be asserted. Reading the *SelectMap Read Data* will return the byte in error. Reading the *Error Location Low* and *High* addresses will return the words containing the address of the error. Finally, reading *Error Word* will return the correct word present in the Flash Memory. Reading the *Error Word* address is also considered a processor acknowledge and causes the module to deassert the interrupt request, clear *Data Ready*, and resume the configuration readback.

C. VHDL CODE

The original VHDL selectmap_rb.vhd module was left intact with two changes. The first modification was disabling all direct access to the PC/104 port. Instead, memory-mapped data registers are loaded with the report data for further access and formatting by the processor. The second modification was adding the flash memory base address option.

In summary, this chapter described the implementation and operation of the X2 Configuration Readback Module. The next chapter describes the Error Reporting Subsystem.

XI. ERROR REPORTING SUBSYSTEM

A. GENERAL DESCRIPTION

The error reporting subsystem takes the syndrome output from the voter modules via the global error bus and makes it available for processor retrieval. A 16 entry queue stores the syndromes until they are retrieved. A delay of at least two clock cycles elapses between the error and processor notification. This delay is insignificant since any correctable error would have already been corrected. The source code for this module is located in Appendix A, Sections G (errorfifo.v) and M (queue1.v).

B. SYNDROME REPORTING BUS

Syndromes are reported from the voter modules to the error reporting subsystem via the global 16-bit errorbus. This bus is configured as a wired-OR network. Bit designations are listed in Table 8.

Bit	**Function**	**Bit**		**Bit**		**Bit**	
0	Error A	4	Spare	8	ID 3	12	ID 7
1	Error B	5	ID 0	9	ID 4	13	ID 8
2	Error C	6	ID 1	10	ID 5	14	ID 9
3	Spare	7	ID 2	11	ID 6	15	Collision Det

Table 8. Syndrome Reporting Bus, Voter Format

C. VOTER CONNECTION

If a voter detects no errors, no syndrome is output to the bus. If a voter detects an error, it drives the bus with its assigned ID and whether input A, B, or C was in error. Since the network is configured as wired-OR and collisions may occur, bit 15 is

designated as a collision detection bit. If the voter is in an error condition and is driving the bus, pin 15 is asserted if bus bits 0-14 are not what the voter is outputting. The collision still occurs and is entered in the queue but is recognized as such.

D. QUEUE

The queue is an eight entry 16-bit queue. The queue module ports are listed in Table 9. It uses the same basic storage structure as the register file except there is only one output bus. Writes take one cycle to propagate through the queue. There are three registers that control queue operation: *first* (4 bits), *store_nxt* (4 bits), and *count* (5 bits). *First* stores the read pointer, *store_nxt* the write pointer, and *count* stores the total number of data elements in the queue. These pointers are advanced depending on the current state and input status. Simultaneous reads and writes are permitted.

Signal	**Bus Size**	**Input/Output**	**Function**
CLK	1	Input	Queue clock
RST	1	Input	Queue reset
ADD	1	Input	Data input to DIN is added to the queue; if count was zero the data is output on the next clock cycle
REMOVE	1	Input	Data is removed from the queue and the next element output to DOUT
DIN	16	Input	Data input for the queue
DOUT	16	Output	Data output for the queue
EMPTY	1	Output	Set if count = 0
NEARLYFULL	1	Output	Set if count = 7
FULL	1	Output	Set if count = 8

Table 9. Queue Module Ports

E. XSOC INTERFACE

The processor interface is implemented as a standard XSOC I/O module. When the queue has data for the processor to read, it outputs that data to *dout* and the empty flag is deasserted. The *Error Data Present* status address as well as the interrupt request lines are simply the inverted output of the empty flag from the queue. Caution should be observed when enabling interrupts to the Error Reporting Module since a configuration error could cause a report on every clock cycle causing constant execution of the interrupt handling routine. When the *Error Data* address is read it also causes a queue remove operation to take place. The address map is shown in Table 10.

Address (Hex)	**Function**
0x00	Error Data, queue remove
0x02	Error Data Present

Table 10. Error Reporting Module Address Map

F. FAULT TOLERANCE IN THE ERROR REPORTING MODULE

The error reporting module is intended to be used as an indication of errors occurring and not to be used for taking immediate action upon receipt of a single error. Considering the cost of making a redundant error reporting bus the decision was made not to implement any fault tolerance on the data portions of the Error Reporting Module. The queue control structure and XSOC I/O interface are fault tolerant, however.

In summary, this chapter described the global error reporting bus and the Error Reporting Module that allows XSOC to retrieve the errors. The next chapter describes the PC/104 Interface Module.

THIS PAGE INTENTIONALLY LEFT BLANK

XII. PC/104 INTERFACE MODULE

A. GENERAL DESCRIPTION

The PC/104 Interface module manages the connection from X1 to the PC/104 processor. Handshaking is done via a memory mapped status register. A 16-entry queue buffers data from X1 to the PC/104 processor board. The PC/104 Interface module xsoc_pc104 was created by translating the VHDL original created by Mindy Surratt into Verilog [15]. Two notable changes were made to the existing code: replacing the Xilinx CoreGen generated queue and adding the XSOC interface. Source code for this module is located in Appendix A, Sections W (xsoc_pc104.v) and K (pc104queue.v).

B. XSOC INTERFACE

The XSOC interface offers I/O mapped operation for writing data and either I/O mapped or interrupt-driven received data. The address map is shown in Table 11.

Address (Hex)	**Function**
0x00	Data (In or Out)
0x02	Output queue full
0x03	Output queue empty
0x04	Output queue nearly full
0x05	Received Data Ready

Table 11. PC/104 Interface Address Map

Data writes to the PC/104 bus are initiated by the processor when the *Data* address is written. The *LD_CE* signal enables an add operation to place the value on the data bus in the queue. Only byte write operations are supported. There are three status bytes that indicate the output queue status. *QUEUE FULL*, indicating that there is no room for additional output data, *QUEUE EMPTY* indicating the queue is empty, and

OUTPUT QUEUE NEARLY FULL, which indicates there is only one free slot remaining in the queue. A non-zero value in these bytes mean the condition is true.

Data reads from the PC/104 bus are accomplished by reading the *Data* address. The *Received Data Ready* address is non-zero when data is ready to be retrieved. The interrupt request line assigned to the PC/104 interface will also be asserted if there is data waiting. There is no ability to disable interrupts in the PC/104 interface module; that task is accomplished in the Interrupt Control module. When the data address is read it changes the status register to acknowledge that XSOC is ready for more data, clears the *Received Data Ready* status byte, and deasserts the interrupt request.

C. QUEUE

The CoreGen queue was replaced with one using a similar structure to the one used in the Error Reporting Interface. The first implementation of the queue was 64 entries as was the original CoreGen part but due to limited FPGA resources it was reduced to 16 entries. This reduction should not pose a problem as the RAM in the microcontroller may be used as additional buffering space. An output register was also added to the Error Reporting Interface queue structure as the CoreGen queue did not output data until a read operation was processed.

Since the PC/104 bus operates asynchronously with X1, the queue had to operate with multiple clock domains. The X1 ~20Mhz clock is considered more than two times faster than the PC/104 data transfer rate so the X1 clock was used to clock queue operations, with only the read operation clocked by the *T_IOREAD_i* signal from the PC/104 bus. The read is captured by a D flip-flop with an asynchronous clear, with the output of the flip-flop connected to the read input of the queue. When the read input to the queue is high, the queue will reset the flip-flop on the next positive edge when it processes the read operation. The clear signal will stay asserted for one system clock cycle. After the clear, the process is ready for another operation.

D. PC/104 INTERFACE

The PC/104 interface code was basically unchanged from the original design by Mindy Surratt. Table 12 lists the signals associated with the PC/104 interface that are currently in use. (The interface is 16 bits but 16 bit transfers have not been implemented.) Note that handshaking signals are active low. In addition to the handshaking signals, the PC/104 computer can request the status register be driven to the data bus. The status register is shown in Table 12. *ALLOW_PROC_READ* indicates to the PC/104 computer that there is data waiting from CFTP. *BLOCK_PROC_WRITE* indicates to the PC/104 computer that the previous byte written has not been processed and CFTP is not ready for more data. *FIFO_ALMOST_FULL* is connected to the queue output of the same name.

Bit	**Function**
0	ALLOW_PROC_READ
1	BLOCK_PROC_WRITE
2	FIFO_ALMOST_FULL

Table 12. PC/104 Status Register

1. Data Reads

The PC/104 signals are shown in Table 13. The addresses supported by the PC/104 interface are shown in Table 14. The two data sources for the CFTP PC/104 interface are the status register and the output of the queue. Data reads begin when the PC/104 computer drives *T_ADDRESS_i* and *T_IOCS_i* goes low. Then *T_IOREAD_i* goes low, either driving the output of the status register or the output register of the queue to the bus. The bus drivers are disabled when *T_IOREAD_i* goes high.

Name	Width	In/Out	Function
T_DATA_io	8	Both	8-bit bidirectional data transfer
T_ADDRESS_i	10	In	Address
T_IOREAD_i	1	In	Read Strobe
T_IOWRITE_i	1	In	Write Strobe
T_IOCS_i	1	In	Address Strobe
T_INTRPT_o	1	Out	Interrupt request to PC/104 computer

Table 13. PC/104 Interface Signals

Address (Hex)	In/Out	Function
0x340	Both	Data Transfer
0x342	Output	Status Register
0x344	Input	INTACK
0x345	Input	INTOFF
0x346	Input	INTON

Table 14. PC/104 Bus Address Map

2. Data Writes

Data writes begin in the same manner as reads with the PC/104 computer driving *T_ADDRESS_i* and *T_IOCS_i* low. Then *T_IOWRITE_i* is driven low to indicate valid data on the bus. Interrupts to notify the PC/104 processor board of data waiting are supported but not currently in use by the PC/104 board software. Specifically, the three interrupt addresses allow the PC/104 board to turn data ready interrupts on, off, or acknowledge a pending interrupt.

E. FAULT TOLERANCE IN THE PC/104 INTERFACE

The PC/104 Interface is not completely fault tolerant. The queue consumes a great deal of FPGA space; making three redundant copies is not possible due to FPGA resource limitations. The data storage element of the queue is not triplicated; all the other structures in the PC/104 interface are.

In summary, this chapter discussed the implementation and use of the PC/104 Interface Module. The next chapter describes the Timestamp Counter and Interrupt Controller Module.

THIS PAGE INTENTIONALLY LEFT BLANK

XIII. TIMESTAMP COUNTER/INTERRUPT CONTROLLER MODULE

A. GENERAL DESCRIPTION

The Timestamp Counter/Interrupt Module has three functions. First, it implements a dual-port timestamp counter. Second, it implements interrupt handling for XSOC. Third, it functions as a periodic timer. Source code for this module is located in Appendix A, Section U (xscounter.v).

When the project was near completion it was noted that the FPGA resources were insufficient to hold the entire design. The counter was modified for 32, 48, or 64 bit operation, selectable via `define statements in the module. The software interface was not changed; reading values for the words that do not exist in the 48 or 32 bit implementation returns a zero value. Reducing the counter size to 48 bits requires no further action as the timer rolls over after approximately 160 days. The 32-bit timer will roll over every 3.6 minutes. This short rollover time will require a software implementation of the high 32 bits of the counter. It is easily achieved using a periodic timer interrupt to ensure activation of the interrupt service routine that then increments the software counter when a rollover is detected.

B. TIMESTAMP COUNTER

The timestamp counter is intended to replace the 64-bit counter present in the X1 control FPGA. The counter in X1 was used for timestamps on data reports sent to the PC/104 processor board via the PC/104 bus.

This counter has two 64-bit read ports, with each port divided into four 16-bit words. The I/O space address and the corresponding functions are shown in Table 15.

Address (Hex)	Function
0x00	Counter Port 1, Least Significant Word
0x02	Counter Port 1, Intermediate Word 1
0x04	Counter Port 1, Intermediate Word 2
0x06	Counter Port 1, Most Significant Word
0x08	Counter Port 2, Least Significant Word
0x0A	Counter Port 2, Intermediate Word 1
0x0C	Counter Port 2, Intermediate Word 2
0x0E	Counter Port 2, Most Significant Word

Table 15. Counter Address Space

When the least significant word of a port is read, it causes the other 48 bits of the counter to be latched into the three most significant words. In this manner all 64 bits of the timer can be read without concern for data values changing between read cycles. It is not necessary to read all words of a particular port; any read on the least significant word causes a reload of the other registers of that port. Two ports were supplied with the intention that the interrupt handler be given port two and applications given port one.

C. INTERRUPT CONTROLLER

The XSOC SoC has a single input pin that expects a one clock cycle pulse on an interrupt request. It does not disable the interrupt signal while it is processing the interrupt handler.

The interrupt controller module is designed to service the relatively low data requirements of the I/O ports on X1 and was therefore made as simple as possible. The higher-rate X2 interface was intended to be the "main" task of the processor and is implemented using busy-waiting for I/O operations.

The interrupt controller does not allow for interrupts to be interrupted and has no interrupt priority capability. There is no provision to signal XSOC which device caused the interrupt. Since there are relatively few I/O devices attached to XSOC and their status registers only require one clock cycle to check (albeit with additional clock cycles used by conditional branches) implementing a more complicated controller would have yielded a minimal improvement. Additionally, the interrupt handler routine can implement a software "priority" by checking the status of the devices in the desired order of importance.

The interrupt controller has eight input-request inputs. Each input has an enable register in the address map. There is also a global enable for all interrupts. They are equal in priority and once an interrupt is being processed other interrupts must wait until the first interrupt has been serviced by the processor.

The states of the interrupt controller are shown in Figure 14. Transitions occur on positive clock edges. On power-on, the controller starts in the idle state. When it receives one or more interrupts, it then goes to the interrupt state for one clock cycle. It is during this state when the processor receives the interrupt signal. It then continues to the Ack Wait state where it waits for an acknowledgement from XSOC that the interrupt has been processed. It is expected that this will be the last instruction executed before returning execution to the interrupted program. Then the delay state is entered where it will remain until the delay timer expires. The delay timer is defined by the INT_DELAY_CLK constant and is currently set to five clock cycles. These clock cycles are intended to allow the *jal* instruction issued at the conclusion of the interrupt handler to complete before processing another interrupt. Once the delay timer expires it reenters the idle state. Note that this process requires the software interrupt handler to clear the interrupt request from the device that desired service.

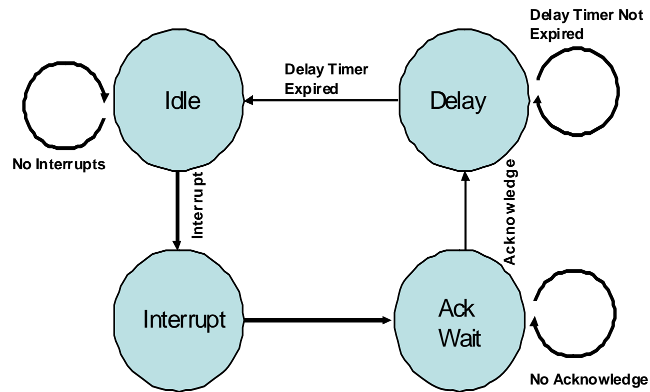

Figure 14. Interrupt Controller States

The address map of the interrupt controller is listed in Table 16. The ports are all write-only. Each address is byte-wide but only bit 0 is used to set the enable value. Writing 0x1 to an address will enable that particular feature. Writing any value to the Interrupt Acknowledge address will send an acknowledge the interrupt controller which will then transition to the delay state.

Address (Hex)	**Function**
0x10	Interrupt Acknowledge
0x11	Global Interrupt Enable
0x12	Interrupt 0 Enable
0x13	Interrupt 1 Enable
0x14	Interrupt 2 Enable
0x15	Interrupt 3 Enable
0x16	Interrupt 4 Enable
0x17	Interrupt 5 Enable
0x18	Interrupt 6 Enable
0x19	Interrupt 7 Enable

Table 16. Interrupt Controller Address Space

D. TIMER

The addition of a timer to the counter required only a small amount of additional logic. The timer module will cause periodic interrupts to occur, with the interrupt period determined by the timer mask. The timer mask is a 16-bit value that corresponds to bits 16-31 of the counter value. The counter bits and timer mask bits are ANDed and then the 16 resulting bits are ORed to form a single output. The result of the AND/OR operation is used to generate an interrupt. Table 17 shows the address map of the timer. The timer mask should only be written with the values corresponding to the timer intervals shown in Table 18. Writing to the Timer Acknowledge address will clear the timer interrupt.

Address (Hex)	Function
0x1A	Timer Mask
0x1C	Timer Acknowledge

Table 17. Timer Address Space

Value (Hex)	Interrupt Period (s)	Value (Hex)	Interrupt Period (s)
0x0001	6.43×10^{-4}	0x0100	1.644
0x0002	1.28×10^{-2}	0x0200	3.28
0x0004	2.57×10^{-2}	0x0400	6.57
0x0008	5.14×10^{-2}	0x0800	13.2
0x0010	0.103	0x1000	26.3
0x0020	0.206	0x2000	52.6
0x0040	0.411	0x4000	105
0x0080	0.822	0x8000	210

Table 18. Timer Mask Values (20.4 MHz System Clock)

This chapter described the design and operation of the Counter and Interrupt Controller Module. It concludes the hardware design section of this thesis. The next chapter describes the process of compiling and loading software into XSOC.

XIV. COMPILING/LOADING XSOC SOFTWARE

A. PREPARING FOR SOFTWARE DEVELOPMENT

The software design and load process consists of four steps: writing C programs, compiling them with a modified version of LCC, converting the output of the C compiler into a Verilog file, and finally compiling the software image into the FPGA load image with ISE.

The first two steps are identical to those detailed by Jan Gray in his "Getting Started with the XSOC Project v0.93" [16]. The XSOC distribution is available at <http://www.fpgacpu.org/xsoc/xsoc-beta-093.zip>. Unzip the distribution and denote the home directory you installed it in as <XSOCHOME>.

Jan Gray modified LCC to work with XSOC. To get the compiler working, install the LCC executables. LCC version 4.1 (now out-of-date and replaced by version 4.2) is still available via ftp in /pub/packages/lcc/lcc41.exe at ftp.cs.princeton.edu. It is a cross-platform C compiler developed by Christopher W. Fraser and David R. Hanson. Install the package and note the installation directory as <LCCHOME>.

Next, copy the files from the <XSOCHOME>/lcc-xr16/bin directory (lcc-xr16.exe, libxr16.s, rcc-xr16.exe, reset.s, and xr16.exe) to the <LCCHOME>/4.1/bin directory. Unless you work in the <LCCHOME>/4.1/bin directory you will need to add that directory to your PATH so Windows will find the executables.

The files reset.s and libxr16.s must be modified for use with our CFTP implementation. Specifically, the stack pointer is set to an out of range address (32K vs. 8K) and there is no need to execute the zero memory routine since the BRAMs are initialized with known values after configuration. Modified libxr16.c and reset.s file contents are included in Appendices D and E. Note that libxr16.c will need to be compiled after modification.

The final tool is the hextov format conversion tool created specifically for this thesis to convert the .hex output of the lcc-xr16 compiler to a Verilog file. This tool converts the ASCII output file that was intended for the XESS development board software loader to a format usable by the CFTP microcontroller.

The block RAMs are initialized through multiple DEFPARAM statements that indicate their startup values. Specifically, each of these DEFPARAM statements contains 256 bits of data. Programs larger than 256 bits will contain multiple DEFPARAM statements per block RAM. The format is a standard Verilog number format, with the four most significant bits as the first hexadecimal character listed. Additionally, since ECC coding is used in the memory subsystem, hextov calculates the correct parity bit information and generates the DEFPARAM statements for those six bits as well.

The tool takes one argument, the name of the input file. This file is assumed to be named .hex; do not add the .hex extension when specifying the filename. The output file will be the same name as the input filename except with a .v extension. For ease of use, copy hextov.exe to the <LCCHOME>/4.1/bin directory.

After generating the .v file, edit the bram4kx22.v file and change the `include entry to the appropriate filename for the file generated above. Then force a recompile of the FPGA.

The CFTP microcontroller was created using Xilinx ISE 9.2i. This was a newer version of the ISE software than had previously been used with CFTP. Additionally, recent projects had not used the ISE Graphical User Interface (GUI) to generate the X1 configuration file. Instead, they had used command-line versions of the ISE tools and a Makefile for the make software building utility to generate the configurations to load into the FPGAs.

The ISE GUI is used to complete the steps up to and including the "Implement Design" process in the ISE processes window. It is not used to generate the programming file. Instead, portions of the make process are duplicated using a small batch file and the bitgen command file. The batch file contains the following two lines:

bitgen -b -w -f bitfile_V1.cmd x1control.ncd
promgen -b -w -u 0 x1control.bit -p bin

The bitgen bitfile_V1.cmd command file contains the bitgen flags necessary for proper operation with CFTP. Configuration files generated without these flags will not function. After the processes in the batch file are complete, a file named x1control.bin containing the finished X1 configuration will be in the project directory.

B. SOFTWARE PROCESS EXAMPLE

The following example assumes the installation tools have been successfully installed. The C file with the program is named "hello.c".

lcc-xr16 -v -c hello.c

lcc-xr16 -v -o hello.hex -lst=hello.lst hello.o

hextov hello

You will now have a <hello.v> file that can be included in the bram4k22.v Verilog source file. Additionally there is a <hello.lst> file that contains a breakdown of memory locations and human-formatted versions of the assembler code they contain. It is useful for debugging but is not required for compilation.

The next chapter summarizes this thesis, presents conclusions, and suggests future improvements to the CFTP design.

THIS PAGE INTENTIONALLY LEFT BLANK

XV. CONCLUSIONS AND RECOMMENDATIONS

This thesis detailed the process of adapting CFTP's X1 control FPGA from one containing fixed-function VHDL blocks to one containing a soft-core microcontroller and software-controlled HDL modules. An existing soft-core System-on-a-Chip RISC computer was extensively modified and enhanced to meet the design goals while existing functional blocks were altered to work with the microcontroller. Lastly, detailed instructions on how to create new I/O modules and how to compile and load software were presented.

A. SUMMARY

The CFTP is a spaceborne platform for testing the effect of SEUs on fault tolerant FPGA designs. It consists of a PC/104 computer board and an experiment board with two FPGAs, with one FPGA designated the experiment and the other the control. The PC/104 computer board and the in-use control FPGA are not fault tolerant. Additionally, the control FPGA is constructed of fixed-function modules that communicate with each other. Both the fixed-function modules in the control FPGA and the experiments in the experiment FPGA are instantiated from HDL using Xilinx ISE CAD tools.

This thesis detailed the process of selecting and modifying an existing microcontroller for use in the CFTP control FPGA. Specifically, XSOC, a soft-core 16-bit RISC-based System-on-a-chip was selected [3]. XSOC was extensively modified to feature TMR, while the memory subsystem was redesigned to utilize Hamming ECC and the block RAMs present in the FPGA. Error detection circuits built into the TMR voters and an Error Reporting I/O module provide indication of error activity.

Existing I/O modules, including the PC/104 interface, the X2 SelectMap configuration, and readback modules were modified to support the new microcontroller interface and were made fault tolerant through use of TMR. A sample I/O module was created with detailed instructions on how to add more I/O modules to the system. A new I/O module was created implementing an interrupt controller, a timestamp counter, and a

programmable timer greatly enhancing the capabilities of the microcontroller. A software tool was created to convert XSOC program binaries into Verilog initialization strings for loading into the FPGA's block RAMs. Finally, a step-by-step guide on creating and loading software in the microcontroller was presented.

B. CONCLUSIONS

A microcontroller-based fault tolerant X1 control FPGA was successfully developed. The microcontroller and PC/104 interface were tested on CFTP hardware. The other I/O modules were tested using a software-based simulation tool.

Functionality was improved due to the flexibility of software controlled modules. The ability to use C and assembly language to program the microcontroller should make implementing future experiments easier as the user only needs to understand the interfaces to the devices rather than the hardware behind them.

There was sufficient room in X1 for a TMR SoC and I/O modules performing the functions of the previous X1 design. However, the design used over 92% of the available slices in the Virtex XQVR-600 FPGA, even after reducing the queue sizes and removing the error notification (but preserving the error correction) capabilities of the I/O device voters. Adding functionality to X1 will come with a cost of reducing existing features.

Using Xilinx Partitions stops global optimization from removing redundant elements in TMR designs. Although not the intended purpose of the feature it works well.

It is possible to make ECC memory nearly as fault tolerant to SEUs as TMR memory with the only weakness being the non-fault-tolerant clock. Such a design uses many voters but offers more memory capacity than TMR when using similar numbers of RAMs.

This thesis should serve as a manual to allow those wishing to use the microcontroller-based design for future experiments. Step-by-step instructions for compiling software and new I/O modules are presented, while the interfaces for I/O modules are defined.

C. RECOMMENDATIONS

These recommendations are listed in order of importance. The first two are necessary to maximize X1's fault tolerance and have a much higher priority than the last three.

1. Implement Configuration Scrubbing on the X1 FPGA

As mentioned in Chapter V, there is no configuration scrubbing on the X1 control FPGA. Without scrubbing, configuration errors will accumulate in X1 until it is manually reset. It is possible that these errors will cause unintended operation. X1 is not completely fault tolerant until scrubbing is enabled and "keeper" circuits are not used.

2. Modify the Code to Eliminate "Keeper" Circuits

As mentioned in Chapter V, when "keeper" circuits are affected by SEUs the configuration errors are not corrected by configuration scrubbing. The VHDL and Verilog code should be modified to eliminate keeper circuit use. If future CFTP projects do not use Virtex I parts this recommendation might not be applicable; time constraints and that possibility were the reason they were not addressed by this thesis. Again, X1 is not completely fault tolerant until these non-correctable configuration errors are eliminated and scrubbing is performed on X1.

3. Make a "Mini-OS" for the CFTP Microcontroller

The microcontroller, although functional, lacks an application that takes full advantage of its features. The "Mini-OS" should include an ISR servicing PC/104 data into CFTP, Error Reporting System, and X2 Configuration Scrubbing System. Additional features should be a sample experiment in X2 and the ability to reconfigure the experiment in X2 on-the-fly.

4. Enable 16-bit Transfers on the PC/104 Bus

Although enabling 16-bit transfers would require changing the handshaking between X1 and the PC/104 processor board and modification to the PC/104 queue and interface, it would double the available bandwidth between the two.

5. Automated Memory Scrubbing

Use the second port of the dual-ported RAMs for a memory scrub. As mentioned in Xilinx XAPP197[14], the second port can be used for this purpose. Relatively simple circuitry can use a counter and small state machine to step through each address, performing a read, correction, and writeback. This functionality would require additional ECC decoders and encoders which might not fit due to lack of available space on the FPGA.

APPENDIX A: VERILOG SOURCE CODE

This appendix contains the Verilog source code used to create the X1 FPGA configuration file. It also contains a sample I/O module file, iotemplate.v, listed in Section H.

A. BITS.V

```
defparam ram0.INIT_00 =
256'h00FC0620000000001C64A8EE3E8480D6581960658002001000003FC05FC0505;
defparam ram1.INIT_00 =
256'h00FC900900001A0A231850083162C2484D813634D850564B000005AAA3AAA203;
defparam ram2.INIT_00 =
256'h00FC110900001A0109642D87A6917A80019806601901543100000333353335O3;
defparam ram3.INIT_00 =
256'h00FC000100004D08201837A6840232180998265099D10141000005C3CBC3CA07;
defparam ram4.INIT_00 =
256'h00FDD8530702DB07FA6940681330C6F85A7D69E5A793333F000001FFF3FFF005;
defparam ram5.INIT_00 =
256'h00FD1C10040288167E0880600801FC1D404101040402000100000800000000001;
defparam ram6.INIT_00 =
256'h0001F5030702DB0181E76F8DF7F9DEF86809A0268093111F0000080000000001;
defparam ram7.INIT_00 =
256'h00FDF3430702DB0000000000000006FD7A79E9E7A793111F0000080000000001;
defparam ram8.INIT_00 =
256'h00FC9431010053102D21695DA9C6A240197865E19797554B000005AAA9AAAA03;
defparam ram9.INIT_00 =
256'h00FC348003011200100010566390A061942050814202000D0000013331333000;
defparam ram10.INIT_00 =
256'h00FCC30700005311212FA32DA645B2802002800A00220013000001C3C1C3C002;
defparam ram11.INIT_00 =
256'h00FCC13700005318106028 0AAC05BAEBB97AE5EB97B7555F000001FC01FC0002;
defparam ram12.INIT_00 =
256'h00FC58C80000380C7D059760595801FA5E8D7A35E8CAAAA1000007FFF4000505;
defparam ram13.INIT_00 =
256'h00FD90890602C90EFE9000E00023F61D64259096424CCCC2000008000800080A;
defparam ram14.INIT_00 =
256'h00FC462000003401014FDF15FFD9B9E209A826A09A822221000007FFF6000505;
defparam ram15.INIT_00 =
256'h00FD108A0502AE0000000000000000051A548552154853333D00000A0007FFF70D;
defparam ram16.INIT_00 =
256'h00FCAD7A020063139BB6E05F11E4F971A34C8D22349CAAD300000DFFF8000801;
defparam ram17.INIT_00 =
256'h0001C1FD0603B40D65EF9BB9A22DC776CB432D3CB4E7021200000A03F7FC070C;
defparam ram18.INIT_00 =
256'h00FC475E00011E0317254F1A803B8074CF8F3E3CF875566D000003FFF5FFF50F;
defparam ram19.INIT_00 =
256'h0000A1B501014111198875A380DB5241A516944A5126666B00000CA5AAA5AA05;
defparam ram20.INIT_00 =
256'h00FC5FDC0300381B92BA609F0AA7F1F4DEFF7BFDEFE0222000003FC0C03FF05;
defparam ram21.INIT_00 =
256'h0001ACF80402841EE2DE345BC3EB7CAE87801E20788EDD96000004A5A2A5A200;
```

B. BRAM4KX22.V

```
`timescale 1ns / 1ps
module bram4kx22(CLK, WE, ADDR, DIN, DOUT);
	input CLK;
	input  [2:0] WE;
	input [47:0] ADDR;
	input [65:0] DIN;
	output [21:0] DOUT;
	wire logic0,logic1;
	assign logic0 = 1'b0;
	assign logic1 = 1'b1;

	wire   [21:0] v_din;
	wire   v_we0;
	wire   v_we1;
	wire   v_we2;
	wire   v_we3;
	wire   v_we4;
	wire   v_we5;
	wire   v_we6;
	wire   v_we7;
	wire   v_we8;
	wire   v_we9;
	wire   v_we10;
	wire   v_we11;
	wire   v_we12;
	wire   v_we13;
	wire   v_we14;
	wire   v_we15;
	wire   v_we16;
	wire   v_we17;
	wire   v_we18;
	wire   v_we19;
	wire   v_we20;
	wire   v_we21;
	wire [11:0] v_addr0;
	wire [11:0] v_addr1;
	wire [11:0] v_addr2;
	wire [11:0] v_addr3;
	wire [11:0] v_addr4;
	wire [11:0] v_addr5;
	wire [11:0] v_addr6;
	wire [11:0] v_addr7;
	wire [11:0] v_addr8;
	wire [11:0] v_addr9;
	wire [11:0] v_addr10;
	wire [11:0] v_addr11;
	wire [11:0] v_addr12;
	wire [11:0] v_addr13;
	wire [11:0] v_addr14;
	wire [11:0] v_addr15;
	wire [11:0] v_addr16;
	wire [11:0] v_addr17;
	wire [11:0] v_addr18;
	wire [11:0] v_addr19;
	wire [11:0] v_addr20;
	wire [11:0] v_addr21;

	voter1n #(1) we0_v(.din(WE), .dout(v_we0));
	voter1n #(1) we1_v(.din(WE), .dout(v_we1));
```

```
        voter1n #(1) we2_v(.din(WE), .dout(v_we2));
        voter1n #(1) we3_v(.din(WE), .dout(v_we3));
        voter1n #(1) we4_v(.din(WE), .dout(v_we4));
        voter1n #(1) we5_v(.din(WE), .dout(v_we5));
        voter1n #(1) we6_v(.din(WE), .dout(v_we6));
        voter1n #(1) we7_v(.din(WE), .dout(v_we7));
        voter1n #(1) we8_v(.din(WE), .dout(v_we8));
        voter1n #(1) we9_v(.din(WE), .dout(v_we9));
        voter1n #(1) we10_v(.din(WE), .dout(v_we10));
        voter1n #(1) we11_v(.din(WE), .dout(v_we11));
        voter1n #(1) we12_v(.din(WE), .dout(v_we12));
        voter1n #(1) we13_v(.din(WE), .dout(v_we13));
        voter1n #(1) we14_v(.din(WE), .dout(v_we14));
        voter1n #(1) we15_v(.din(WE), .dout(v_we15));
        voter1n #(1) we16_v(.din(WE), .dout(v_we16));
        voter1n #(1) we17_v(.din(WE), .dout(v_we17));
        voter1n #(1) we18_v(.din(WE), .dout(v_we18));
        voter1n #(1) we19_v(.din(WE), .dout(v_we19));
        voter1n #(1) we20_v(.din(WE), .dout(v_we20));
        voter1n #(1) we21_v(.din(WE), .dout(v_we21));

        voter1n #(12) addr0_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr0));
        voter1n #(12) addr1_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr1));
        voter1n #(12) addr2_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr2));
        voter1n #(12) addr3_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr3));
        voter1n #(12) addr4_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr4));
        voter1n #(12) addr5_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr5));
        voter1n #(12) addr6_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr6));
        voter1n #(12) addr7_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr7));
        voter1n #(12) addr8_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr8));
        voter1n #(12) addr9_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr9));
        voter1n #(12) addr10_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr10));
        voter1n #(12) addr11_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr11));
        voter1n #(12) addr12_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr12));
        voter1n #(12) addr13_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr13));
        voter1n #(12) addr14_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr14));
        voter1n #(12) addr15_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr15));
        voter1n #(12) addr16_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr16));
        voter1n #(12) addr17_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr17));
        voter1n #(12) addr18_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr18));
        voter1n #(12) addr19_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr19));
```

```
	voter1n #(12) addr20_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr20));
	voter1n #(12) addr21_v(.din({ADDR[44:33], ADDR[28:17], ADDR[12:1]}),
.dout(v_addr21));

	voter1n #(22) din_v(.din(DIN), .dout(v_din));

	RAMB4_S1 ram0(.WE(v_we0), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr0), .DI(v_din[0]), .DO(DOUT[0]));
	RAMB4_S1 ram1(.WE(v_we1), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr1), .DI(v_din[1]), .DO(DOUT[1]));
	RAMB4_S1 ram2(.WE(v_we2), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr2), .DI(v_din[2]), .DO(DOUT[2]));
	RAMB4_S1 ram3(.WE(v_we3), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr3), .DI(v_din[3]), .DO(DOUT[3]));
	RAMB4_S1 ram4(.WE(v_we4), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr4), .DI(v_din[4]), .DO(DOUT[4]));
	RAMB4_S1 ram5(.WE(v_we5), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr5), .DI(v_din[5]), .DO(DOUT[5]));
	RAMB4_S1 ram6(.WE(v_we6), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr6), .DI(v_din[6]), .DO(DOUT[6]));
	RAMB4_S1 ram7(.WE(v_we7), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr7), .DI(v_din[7]), .DO(DOUT[7]));
	RAMB4_S1 ram8(.WE(v_we8), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr8), .DI(v_din[8]), .DO(DOUT[8]));
	RAMB4_S1 ram9(.WE(v_we9), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr9), .DI(v_din[9]), .DO(DOUT[9]));
	RAMB4_S1 ram10(.WE(v_we10), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr10), .DI(v_din[10]), .DO(DOUT[10]));
	RAMB4_S1 ram11(.WE(v_we11), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr11), .DI(v_din[11]), .DO(DOUT[11]));
	RAMB4_S1 ram12(.WE(v_we12), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr12), .DI(v_din[12]), .DO(DOUT[12]));
	RAMB4_S1 ram13(.WE(v_we13), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr13), .DI(v_din[13]), .DO(DOUT[13]));
	RAMB4_S1 ram14(.WE(v_we14), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr14), .DI(v_din[14]), .DO(DOUT[14]));
	RAMB4_S1 ram15(.WE(v_we15), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr15), .DI(v_din[15]), .DO(DOUT[15]));
	RAMB4_S1 ram16(.WE(v_we16), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr16), .DI(v_din[16]), .DO(DOUT[16]));
	RAMB4_S1 ram17(.WE(v_we17), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr17), .DI(v_din[17]), .DO(DOUT[17]));
	RAMB4_S1 ram18(.WE(v_we18), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr18), .DI(v_din[18]), .DO(DOUT[18]));
	RAMB4_S1 ram19(.WE(v_we19), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr19), .DI(v_din[19]), .DO(DOUT[19]));
	RAMB4_S1 ram20(.WE(v_we20), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr20), .DI(v_din[20]), .DO(DOUT[20]));
	RAMB4_S1 ram21(.WE(v_we21), .EN(logic1), .RST(logic0), .CLK(CLK),
.ADDR(v_addr21), .DI(v_din[21]), .DO(DOUT[21]));
	`include "bits.v"
endmodule

module decode5(din, dout);
	input [4:0] din;
	output [21:0] dout;

	reg [21:0] dout;

	always @(din) begin
		case (din)
```

```
                    5'h0 : dout <= 22'h000000;
                    5'h1 : dout <= 22'h000000;
                    5'h2 : dout <= 22'h000000;
                    5'h3 : dout <= 22'h000001;
                    5'h4 : dout <= 22'h000000;
                    5'h5 : dout <= 22'h000002;
                    5'h6 : dout <= 22'h000004;
                    5'h7 : dout <= 22'h000008;
                    5'h8 : dout <= 22'h000000;
                    5'h9 : dout <= 22'h000010;
                    5'hA : dout <= 22'h000020;
                    5'hB : dout <= 22'h000040;
                    5'hC : dout <= 22'h000080;
                    5'hD : dout <= 22'h000100;
                    5'hE : dout <= 22'h000200;
                    5'hF : dout <= 22'h000400;
                    5'h10 : dout <= 22'h000000;
                    5'h11 : dout <= 22'h000800;
                    5'h12 : dout <= 22'h001000;
                    5'h13 : dout <= 22'h002000;
                    5'h14 : dout <= 22'h004000;
                    5'h15 : dout <= 22'h008000;
                    5'h16 : dout <= 22'h000000;
                    5'h17 : dout <= 22'h000000;
                    5'h18 : dout <= 22'h000000;
                    5'h19 : dout <= 22'h000000;
                    5'h1A : dout <= 22'h000000;
                    5'h1B : dout <= 22'h000000;
                    5'h1C : dout <= 22'h000000;
                    5'h1D : dout <= 22'h000000;
                    5'h1E : dout <= 22'h000000;
                    5'h1F : dout <= 22'h000000;
                          default : dout <= 22'h000000;
             endcase
      end
endmodule

module eccencode(din, dout);
    input [15:0] din;
    output [21:0] dout;

      assign dout[15:0] = din[15:0];
      // check group 1
      assign dout[16] = din[0] ^ din[1] ^ din[3] ^ din[4] ^ din[6] ^ din[8] ^
din[10] ^ din[11] ^ din[13] ^ din[15];
      // check group 2
      assign dout[17] = din[0] ^ din[2] ^ din[3] ^ din[5] ^ din[6] ^ din[9] ^
din[10] ^ din[12] ^ din[13];
      // check group 4
      assign dout[18] = din[1] ^ din[2] ^ din[3] ^ din[7] ^ din[8] ^ din[9] ^
din[10] ^ din[14] ^ din[15];
      // check group 8
      assign dout[19] = ^din[10:4];
      // check group 16
      assign dout[20] = ^din[15:11];
      // overall check
      //assign dout[21] = (^din[15:0]) ^ (^dout[20:16]);
      assign dout[21] = 1'b0;

endmodule

module eccdecode(din, dout, syndrome);
```

```
        input [21:0] din;
        output [15:0] dout;
        output [5:0] syndrome;

    wire [5:0] syndrome;
    wire [21:0] correct;

    // check group 1
    assign syndrome[0] = din[16] ^ din[0] ^ din[1] ^ din[3] ^ din[4] ^ din[6]
^ din[8] ^ din[10] ^ din[11] ^ din[13] ^ din[15];
    // check group 2
    assign syndrome[1] = din[17] ^ din[0] ^ din[2] ^ din[3] ^ din[5] ^ din[6]
^ din[9] ^ din[10] ^ din[12] ^ din[13];
    // check group 4
    assign syndrome[2] = din[18] ^ din[1] ^ din[2] ^ din[3] ^ din[7] ^ din[8]
^ din[9] ^ din[10] ^ din[14] ^ din[15];
    // check group 8
    assign syndrome[3] = din[19] ^ (^din[10:4]);
    // check group 16
    assign syndrome[4] = din[20] ^ (^din[15:11]);

    assign syndrome[5] = ^din;

    decode5 eccdecode(.din(syndrome[4:0]),.dout(correct));

    assign dout = din[15:0] ^ correct[15:0];

endmodule
```

C. BRAM_ECC.V

```
`timescale 1ns / 1ps
`include "voterids.v"

module bram_ecc(clk, rst, sub, slb, rub, uxd_t, lxd_t, memdta,
        mem_addr, we, d, xd, errorbus, addr_nxt, mem_ce);

    input                       clk;
    input                       rst;
    input  [2:0]         sub;
    input  [2:0]         slb;
    input  [2:0]         rub;
    input [2:0]  uxd_t;
    input [2:0]  lxd_t;
    input  [47:0]        memdta;
    output [47:0]        mem_addr;
    input  [2:0]         we;
    inout  [47:0]        d;
    output [47:0]        xd;
    input  [47:0] addr_nxt;
    input [2:0]          mem_ce;
    inout  [15:0] errorbus;

    wire   [17:0] syndrome;
    wor    [15:0] errorbus;
    wire   [65:0]        eccdin;
    wire   [21:0]        eccdout;
    tri    [47:0]  d;
    wire   [47:0]        mem_out;
```

```
	reg	[47:0] xa;
	wire	[15:0] v_addr_nxt0, v_addr_nxt1, v_addr_nxt2;
	wire			aerror_nxt, berror_nxt, cerror_nxt;
	reg			aerror, berror, cerror;


	voter1nsyn #(16)	addr_nxt_v0 (.din(addr_nxt), .dout(v_addr_nxt0),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`BRAM_ADDR_NXT0_V}));
	voter1nsyn #(16)	addr_nxt_v1 (.din(addr_nxt), .dout(v_addr_nxt1),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`BRAM_ADDR_NXT1_V}));
	voter1nsyn #(16)	addr_nxt_v2 (.din(addr_nxt), .dout(v_addr_nxt2),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`BRAM_ADDR_NXT2_V}));

	assign mem_addr[15:0] = mem_ce[0] ? addr_nxt[15:0] : xa[15:0];
	assign mem_addr[31:16] = mem_ce[1] ? addr_nxt[31:16] : xa[31:16];
	assign mem_addr[47:32] = mem_ce[2] ? addr_nxt[47:32] : xa[47:32];

	always @(posedge clk or posedge rst) begin
			if (rst) begin
				xa <= 0;
			end
			else begin
				if (mem_ce[0]) begin
					xa[15:0] <= v_addr_nxt0;
				end
				if (mem_ce[1])	begin
					xa[31:16] <= v_addr_nxt1;
				end
				if (mem_ce[2]) begin
					xa[47:32] <= v_addr_nxt2;
				end
			end
	end

	// code lifted from voter module
	assign aerror_nxt= |(syndrome[5:0]);
	assign berror_nxt= |(syndrome[11:6]);
	assign cerror_nxt= |(syndrome[17:12]);

	always @(posedge clk or posedge rst) begin
		if (rst) begin
			aerror <= 0;
			berror <= 0;
			cerror <= 0;
		end
		else begin
			aerror<= aerror_nxt;
			berror<= berror_nxt;
			cerror<= cerror_nxt;
		end
	end

	assign errorbus[14:5] = ((aerror) | (berror) | (cerror)) ?
`BRAM_SYNDROME_ERR : 10'd0;
	assign errorbus[4:3] = 2'd0;
	assign errorbus[2:0] = ((aerror) | (berror) | (cerror)) ?
{(cerror),(berror),(aerror)} : 3'd0;
	// collision detect
	assign errorbus[15] = ((aerror) | (berror) | (cerror)) ?
((errorbus[14:0]==
			({`BRAM_SYNDROME_ERR,2'd0,(cerror),(berror),(aerror)})) ?
1'b0 : 1'b1) : 1'b0;
```

```
	memio memio1 (.sub(sub[0]), .slb(slb[0]), .rub(rub[0]),
		.din(memdta[15:0]), .eccdout(eccdout), .mem_out(mem_out[15:0]),
		.xd(xd[15:0]), .eccdin(eccdin[21:0]), .syndrome(syndrome[5:0]));


	memio memio2 (.sub(sub[1]), .slb(slb[1]), .rub(rub[1]),
		.din(memdta[31:16]), .eccdout(eccdout), .mem_out(mem_out[31:16]),
		.xd(xd[31:16]), .eccdin(eccdin[43:22]),
.syndrome(syndrome[11:6]));

	memio memio3 (.sub(sub[2]), .slb(slb[2]), .rub(rub[2]),
		.din(memdta[47:32]), .eccdout(eccdout), .mem_out(mem_out[47:32]),
		.xd(xd[47:32]), .eccdin(eccdin[65:44]),
.syndrome(syndrome[17:12]));

	bram4kx22 mem1 (.CLK(clk), .WE(we), .ADDR(mem_addr),
		.DIN(eccdin), .DOUT(eccdout));

	assign d[7:0] = lxd_t[0] ? 8'bz: mem_out[7:0];
	assign d[15:8] = uxd_t[0] ? 8'bz: mem_out[15:8];
	assign d[23:16] = lxd_t[1] ? 8'bz: mem_out[23:16];
	assign d[31:24] = uxd_t[1] ? 8'bz: mem_out[31:24];
	assign d[39:32] = lxd_t[2] ? 8'bz: mem_out[39:32];
	assign d[47:40] = uxd_t[2] ? 8'bz: mem_out[47:40];


endmodule
```

D. CLOCKDIV.V

```
// module to derive a ~20Mhz system clock from the 51Mhz PC/104 clock
`timescale 1ns / 1ps
module clock_div(ACLK, CLKDIV, RST);
input ACLK;
input RST;
output CLKDIV;

IBUFG CLK_ibufg_A (.I (ACLK), .O(ACLK_ibufg));

BUFG DIVCLK_bufg (.I (ACLK_div2), .O (CLKDIV));

BUFG ACLK_FB (.I (ACLK_0), .O (ACLK_feedback));

CLKDLL ACLK_dll_div // /2.5 clock
(.CLKIN(ACLK_ibufg),
.CLKFB(ACLK_feedback),
.RST(RST),
.CLK2X(),
.CLK0(ACLK_0),
.CLK90(),
.CLK180(),
.CLK270(),
.CLKDV(ACLK_div2),
.LOCKED()
);

defparam ACLK_dll_div.CLKDV_DIVIDE=2.5;

endmodule
```

E. CTRL.V

```
/* ctrl.v -- xr16 control unit synthesizable Verilog model
 *
 * Copyright (C) 1999, 2000, Gray Research LLC.  All rights reserved.
 * The contents of this file are subject to the XSOC License Agreement;
 * you may not use this file except in compliance with this Agreement.
 * See the LICENSE file.
 *
 * $Header: /dist/xsocv/ctrl.v 8     4/06/00 10:55a Jan $
 * $Log: /dist/xsocv/ctrl.v $
 *
 * 8     4/06/00 10:55a Jan
 * polish
 */
 /* last modified by David Dwiggins, 24 August 2008.  Adapted to use
 *  TMR/ECC functionality for fault-tolerance.  Also adapted to reload each
register
*      each clock cycle */
// original code by Jan Gray included at end of file

`timescale 1ns / 1ps
`include "voterids.v"

// Instruction fields, ref.: The xr16 Specifications (doc/xspecs.pdf).

// Opcodes (IR[15:12])
`define ADD         4'h0
`define SUB         4'h1
`define      ADDI   4'h2
`define      RR            4'h3
`define RI          4'h4
`define LW          4'h5
`define LB          4'h6
`define SW          4'h8
`define SB          4'h9
`define JAL         4'hA
`define Bcond 4'hB
`define CALL  4'hC
`define IMM         4'hD

// Functions (IR[7:4])
`define AND         4'h0
`define OR          4'h1
`define XOR         4'h2
`define ANDN  4'h3
`define ADC         4'h4
`define SBC         4'h5
`define SRL         4'h6
`define SRA         4'h7
`define      SLL           4'h8

// Conditional branches (IR[11:8])
`define BR          4'h0
`define BRN         4'h1
`define      BEQ           4'h2
`define      BNE           4'h3
`define      BC            4'h4
`define      BNC           4'h5
`define      BV            4'h6
`define      BNV           4'h7
```

```
`define      BLT         4'h8
`define      BGE         4'h9
`define      BLE         4'hA
`define BGT         4'hB
`define      BLTU   4'hC
`define      BGEU   4'hD
`define      BLEU   4'hE
`define      BGTU   4'hF

`define INT         16'hAE01     // int ::= jal r14,10(r0)

module control(
       clk, rst, rdy, int_req, dma_req, zerodma, insn, a15, z, n, co, v,
       mem_ce, word_nxt, read_nxt, dbus_nxt, dma,
       rf_we, rna, rnb, rdest,
       fwd, imm, sextimm4, zextimm4, wordimm4, imm12, pipe_ce, b15_4_ce,
       add, ci, logicop, sri,
       sum_t, logic_t, shl_t, shr_t, zeroext_t, ret_t,
       branch, brdisp, selpc, zeropc, dmapc, pc_ce, ret_ce,
       in_add, out_add, in_ci, out_ci, in_branch, out_branch,
       in_sum_t, out_sum_t, in_logic_t, out_logic_t, in_shl_t, out_shl_t,
       in_shr_t, out_shr_t, in_zeroext_t, out_zeroext_t, in_ret_t, out_ret_t,
       in_if_ir, out_if_ir, in_ir, out_ir, in_ex_ir, out_ex_ir, in_ex_call,
out_ex_call,
       in_ex_st, out_ex_st, in_ifetch, out_ifetch, in_dma, out_dma,
in_sync_reset, out_sync_reset,
       in_dc_annul, out_dc_annul, in_ex_annul, out_ex_annul, in_int_pend,
out_int_pend,
       in_dc_int, out_dc_int, in_dma_pend, out_dma_pend, in_zero_pend,
out_zero_pend,
       inum, errorbus);

       parameter         W      = 16;  // data word width
       parameter         N      = W-1; // msb index
       parameter         IW     = 16;  // instruction word width
       parameter         IN     = 15;  // instruction word msb index

       // ports
       input             clk;          // global clock
       input             rst;          // global async reset
       input             [2:0]  rdy;          // current memory access is ready
       input                          int_req;      // interrupt request
       input                          dma_req;      // DMA request
       input                          zerodma;      // zero DMA counter
request
       input             [15:0] insn;         // new instruction word
       input                          a15;        // A operand msb
       input                          z;                // zero result
condition code
       input                          n;                // negative result
condition code
       input                          co;               // carry-out result
codition code
       input                          v;                // oVerflow result
condition code
       input             [2:0] in_add, in_ci, in_branch, in_sum_t, in_logic_t;
       input             [2:0] in_shl_t, in_shr_t, in_zeroext_t, in_ret_t;
       input             [47:0] in_if_ir, in_ir, in_ex_ir;
       input             [2:0] in_ex_call, in_ex_st, in_ifetch, in_dma,
in_sync_reset;
       input             [2:0] in_dc_annul, in_ex_annul, in_int_pend,
in_dc_int, in_dma_pend;
```

```
	input		[2:0] in_zero_pend;
	input		[1:0]	inum;
	output		mem_ce;		// memory access clock enable
	output		word_nxt;	// next access is word wide
	output		read_nxt;	// next access is read
	output		dbus_nxt;	// next access uses on-chip data bus
	output		dma;		// current access is a DMA transfer
	output		rf_we;		// register file write enable
	output [3:0]	rna;		// register file port A register number
	output [3:0]	rnb;		// register file port B register number
	output	[3:0]	rdest;	// register file write number
	output		fwd;		// forward result bus into A operand
register
	output [11:0] imm;		// 12-bit immediate field
	output		sextimm4;	// sign-extend 4-bit immediate operand
	output		zextimm4;	// zero-extend 4-bit immediate operand
	output		wordimm4;	// word-offset 4-bit immediate operand
	output		imm12;		// 12-bit immediate operand
	output		pipe_ce;	// pipeline clock enable
	output		b15_4_ce;	// b[15:4] clock enable
	output		add;		// 1 => A + B; 0 => A - B
	output		ci;			// carry-in
	output [1:0]	logicop;	// logic unit opcode
	output		sri;		// shift right msb input
	output		sum_t;		// active low adder output enable
	output		logic_t;	// active low logic unit output enable
	output		shl_t;		// active low shift left output enable
	output		shr_t;		// active low shift right output enable
	output		zeroext_t;	// active low zero-extension output
enable
	output		ret_t;		// active low return address output
enable
	output		branch;			// branch taken
	output [7:0]	brdisp;		// 8-bit branch displacement
	output		selpc;		// address mux selects next PC
	output		zeropc;			// force next PC to 0
	output		dmapc;		// use DMA register in PC register file
	output		pc_ce;		// PC clock enable
	output		ret_ce;			// return address clock enable
	output		out_add, out_ci, out_branch, out_sum_t, out_logic_t;
	output		out_shl_t, out_shr_t, out_zeroext_t, out_ret_t;
	output		[15:0] out_if_ir, out_ir, out_ex_ir;
	output		out_ex_call, out_ex_st, out_ifetch, out_dma,
out_sync_reset;
	output		out_dc_annul, out_ex_annul, out_int_pend, out_dc_int,
out_dma_pend;
	output		out_zero_pend;
	inout			[15:0] errorbus;

	reg			add, ci, branch;
	reg			sum_t, logic_t, shl_t, shr_t, zeroext_t,
ret_t;

	// locals
	reg [IN:0]	if_ir, ir, ex_ir;	// instruction registers

	// IR fields
	wire [3:0]	op		= ir[15:12];	// opcode
	wire [3:0]	rd		= ir[11:8];		// destination register
	wire [3:0]	ra		= ir[7:4];		// source register A
	wire [3:0]	rb		= ir[3:0];		// source register B
	wire [3:0]	cond	= ir[11:8];		// branch condition
```

```
	wire [3:0]	fn		= ir[7:4];		// rr or ri function sub-
opcode
	wire [3:0]	ex_op	= ex_ir[15:12];		// EX stage opcode
	wire [3:0]	ex_rd	= ex_ir[11:8];		// EX stage destination register
	wire [3:0]	ex_fn	= ex_ir[7:4];// EX stage function sub-opcode

	wire	v_rdy;

	wire			v_add, v_ci, v_branch, v_sum_t, v_logic_t;
	wire			v_shl_t, v_shr_t, v_zeroext_t, v_ret_t;
	wire			[15:0] v_if_ir, v_ir, v_ex_ir;
	wire			v_ex_call, v_ex_st, v_ifetch, v_dma, v_sync_reset;
	wire			v_dc_annul, v_ex_annul, v_int_pend, v_dc_int,
v_dma_pend;
	wire			v_zero_pend;

	wor	[15:0] errorbus;

	voter1nsyn #(1)		rdy_v (.din(rdy), .dout(v_rdy), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID({`CO_RDY_V}));

	voter1nsyn #(1)		add_v (.din(in_add), .dout(v_add),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_ADD_V}));
	voter1nsyn #(1)		ci_v (.din(in_ci), .dout(v_ci), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID({`CO_CI_V}));
	voter1nsyn #(1)		branch_v (.din(in_branch), .dout(v_branch),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_BRANCH_V}));
	voter1nsyn #(1)		sum_t_v (.din(in_sum_t), .dout(v_sum_t),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_SUM_T_V}));
	voter1nsyn #(1)		logic_t_v (.din(in_logic_t), .dout(v_logic_t),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_LOGIC_T_V}));
	voter1nsyn #(1)		shl_t_v (.din(in_shl_t), .dout(v_shl_t),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_SHL_T_V}));
	voter1nsyn #(1)		shr_t_v (.din(in_shr_t), .dout(v_shr_t),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_SHR_T_V}));
	voter1nsyn #(1)		zeroext_t_v (.din(in_zeroext_t), .dout(v_zeroext_t),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_ZEROEXT_T_V}));
	voter1nsyn #(1)		ret_t_v (.din(in_ret_t), .dout(v_ret_t),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_RET_T_V}));
	voter1nsyn #(16)	if_ir_v (.din(in_if_ir), .dout(v_if_ir),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_IF_IR_V}));
	voter1nsyn #(16)	ir_v (.din(in_ir), .dout(v_ir), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID({`CO_IR_V}));
	voter1nsyn #(16)	ex_ir_v (.din(in_ex_ir), .dout(v_ex_ir),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_EX_IR_V}));
	voter1nsyn #(1)		ex_call_v (.din(in_ex_call), .dout(v_ex_call),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_EX_CALL_V}));
	voter1nsyn #(1)		ex_st_v (.din(in_ex_st), .dout(v_ex_st),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_EX_ST_V}));
	voter1nsyn #(1)		ifetch_v (.din(in_ifetch), .dout(v_ifetch),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_IFETCH_V}));
	voter1nsyn #(1)		dma_v (.din(in_dma), .dout(v_dma),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_DMA_V}));
	voter1nsyn #(1)		sync_reset_v (.din(in_sync_reset),
.dout(v_sync_reset), .errorbus(errorbus), .clk(clk), .rst(rst),
.ID({`CO_SYNC_RESET_V}));
	voter1nsyn #(1)		dc_annul_v (.din(in_dc_annul), .dout(v_dc_annul),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_DC_ANNUL_V}));
	voter1nsyn #(1)		ex_annul_v (.din(in_ex_annul), .dout(v_ex_annul),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_EX_ANNUL_V}));
	voter1nsyn #(1)		int_pend_v (.din(in_int_pend), .dout(v_int_pend),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_INT_PEND_V}));
```

```
	voter1nsyn #(1)	dc_int_v (.din(in_dc_int), .dout(v_dc_int),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_DC_INT_V}));
	voter1nsyn #(1)	dma_pend_v (.din(in_dma_pend), .dout(v_dma_pend),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_DMA_PEND_V}));
	voter1nsyn #(1)	zero_pend_v (.din(in_zero_pend), .dout(v_zero_pend),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`CO_ZERO_PEND_V}));

	assign errorbus = 16'b0;

	assign	imm	= ir[11:0];	// 12-bit immediate field
	assign	brdisp = ex_ir[7:0]; // 8-bit branch displacement

	// IF stage instruction decoding
	assign imm12 = op==`CALL || op==`IMM;
	assign sextimm4	= op==`ADDI || op==`RI;
	assign zextimm4	= op==`LB || op==`SB;
	assign wordimm4 = op==`LW || op==`SW || op==`JAL;
	wire addsub	= op==`ADD || op==`SUB || op==`ADDI;
	wire call	= op==`CALL;
	wire st			= op==`SW || op==`SB;
	wire rrri	= op==`RR || op == `RI;
	wire adcsbc	= rrri && (fn==`ADC || fn==`SBC);
	wire dcintinh = op==`ADD || op==`SUB || op==`ADDI || op==`Bcond ||
				op==`CALL || op==`IMM || adcsbc;
	wire sub	= op==`SUB || (rrri && (fn==`SBC));

	// EX stage decoding
	wire ex_ldst = ex_op==`LW || ex_op==`LB || ex_op==`SW || ex_op==`SB;
	wire ex_lbsb = ex_op==`LB || ex_op==`SB;
	wire ex_results	= ex_op==`ADD || ex_op==`SUB || ex_op==`ADDI ||
				ex_op==`RR || ex_op==`RI ||
				ex_op==`LW || ex_op==`LB || ex_op==`JAL ||
ex_op==`CALL;
	wire ex_jump = (ex_op==`JAL || ex_op==`CALL);
	reg  ex_call; // EX stage call
	reg  ex_st;	// EX stage store

	// memory access FSM states -- each access is an instruction fetch,
	// unless DMA or load/store is pending
	reg  ifetch; // "current access is insn fetch"
	reg  dma;	// "current access is DMA"
			// "current access is load/store" ::=
~(ifetch|dma)
	// annul FSM states
	reg  sync_reset; // synchronous memory FSM reset
	reg  dc_annul;	// annul DC stage instruction
	reg  ex_annul;	// annul EX stage instruction

	// interrupt state
	reg  int_pend;	// interrupt is pending
	reg  dc_int; // DC stage instruction is int
	wire if_int; // IF stage 'interrupt in progress'

	// DMA state
	reg	dma_pend;	// DMA is pending
	reg  zero_pend;	// zero (reset) DMA is pending

	// memory access FSM transitions:
	// case (state)
	//   IF:  state_nxt = dma_pend ? DMA : ldst_pend ? LS : IF;
	//   DMA: state_nxt =                  ldst_pend ? LS : IF;
	//   LS:  state_nxt =                                   IF;
```

```
// endcase
wire ldst_pend      = ex_ldst & ~ex_annul;
wire if_nxt         = ifetch&~dma_pend&~ldst_pend | dma&~ldst_pend |
                            ~ifetch&~dma;
wire dma_nxt  = ifetch&dma_pend;
wire ldst_nxt = ifetch&~dma_pend&ldst_pend | dma&ldst_pend;

wire jump           = ex_jump && ~ex_annul;
wire exannul_nxt= sync_reset | branch | jump | dc_annul;

// FSM output decodes
assign mem_ce = v_rdy;
assign pipe_ce      = v_rdy&if_nxt;
assign pc_ce  = v_rdy&(if_nxt|dma_nxt);
assign ret_ce = v_rdy&if_nxt&~dc_int;
assign word_nxt     = ~(ldst_nxt&ex_lbsb);
assign read_nxt     = ~(ldst_nxt&ex_st);
assign dbus_nxt     = ldst_nxt;
assign dmapc  = dma_nxt;
assign selpc  = if_nxt&~jump | dma_nxt;
assign zeropc = (zero_pend&dma_nxt) | sync_reset;
assign if_int = int_pend&~branch&~jump&~dcintinh;


// IR clocking
assign out_if_ir = (ifetch && v_rdy) ? insn : if_ir;
assign out_ir = pipe_ce ? ((if_int) ? `INT : (ifetch) ? insn : if_ir) :
ir;
assign out_ex_ir = pipe_ce ? ir : ex_ir;

always @(posedge clk or posedge rst) begin
       if (rst) begin
              if_ir <=0;
              ir <= 0;
              ex_ir <= 0;
       end
       else begin
              if_ir <= v_if_ir;
              ir <= v_ir;
              ex_ir <= v_ex_ir;
       end
end


// instruction decode clocking
assign out_ex_call = pipe_ce ? call : ex_call;
assign out_ex_st = pipe_ce ? st : ex_st;

always @(posedge clk or posedge rst) begin
       if (rst) begin
              ex_call <= 0;
              ex_st <= 0;
       end
       else begin
              ex_call <= v_ex_call;
              ex_st <= v_ex_st;
       end
end

// memory access FSM clocking

assign out_ifetch = v_rdy ? if_nxt : ifetch;
```

```
	assign out_dma = v_rdy ? dma_nxt : dma;

	always @(posedge clk or posedge rst) begin
		if (rst) begin
			ifetch <= 1;
			dma <= 0;
		end
		else begin
		ifetch <= v_ifetch;
		dma <= v_dma;
		end
	end

	// annul states clocking

	assign out_sync_reset = v_rdy ? 1'b0 : sync_reset;
	assign out_dc_annul = pipe_ce ? (sync_reset | branch | jump) : dc_annul;
	assign out_ex_annul = pipe_ce ? exannul_nxt : ex_annul;

	always @(posedge clk or posedge rst) begin
		if (rst) begin
			sync_reset <= 1;
			dc_annul <= 1;
			ex_annul <= 1;
		end
		else begin
			sync_reset <= v_sync_reset;
			dc_annul <= v_dc_annul;
			ex_annul <= v_ex_annul;
		end
	end

	// DMA request clocking
	assign out_dma_pend = dma_req ? 1'b1 : (dma ? 1'b0 : dma_pend);
	assign out_zero_pend = zerodma ? 1'b1 : (dma ? 1'b0 : zero_pend);

	always @(posedge clk or posedge rst) begin
		if (rst) begin
			dma_pend <= 0;
			zero_pend <= 0;
		end
		else begin
			dma_pend <= v_dma_pend;
			zero_pend <= v_zero_pend;
		end
	end

	// interrupt request clocking
	assign out_int_pend = int_req ? 1'b1 : ((if_int && pipe_ce) ? 1'b0 :
int_pend);
	assign out_dc_int = pipe_ce ? if_int : dc_int;

	always @(posedge clk or posedge rst) begin
		if (rst) begin
			int_pend <= 0;
			dc_int <= 0;
		end
		else begin
			int_pend <= v_int_pend;
			dc_int <= v_dc_int;
		end
	end
```

```
	// DC stage operand selection
	wire [3:0]	rsrc	= call ? 0 : rrri ? rd : ra;
	wire [3:0]	rdest	= ex_call ? 15 : ex_rd;
//	assign		rna		= clk ? rsrc : rdest;
//	assign		rnb		= clk ? (st ? rd : rb) : (ex_call ? 15 :
ex_rd);
	assign		rna		= rsrc;
	assign		rnb		= (st ? rd : rb);
	assign		rf_we	= ex_results & pipe_ce & ~ex_annul & (rdest != 0);
	assign		fwd		= (rsrc==rdest) & (rdest!=0) && ex_results &
~ex_annul;
	assign		b15_4_ce= pipe_ce && ~(ex_op==`IMM && ~ex_annul);

	// DC stage conditional branches
	reg taken;

	always @(cond or z or co or v or n) begin
			case (cond)
			`BR:	taken = 1;
			`BRN:	taken = 0;
			`BEQ:	taken = z;
			`BNE:	taken = ~z;
			`BC:	taken = co;
			`BNC:	taken = ~co;
			`BV:	taken = v;
			`BNV:	taken = ~v;
			`BLT:	taken = n^v;
			`BGE:	taken = ~(n^v);
			`BLE:	taken = (n^v)|z;
			`BGT:	taken = ~((n^v)|z);
			`BLTU: taken = ~z&~co;
			`BGEU: taken = z|co;
			`BLEU: taken = z|~co;
			`BGTU: taken = ~z&co;
	endcase
	end

	assign out_branch = pipe_ce ? (op==`Bcond && taken && ~exannul_nxt) :
branch;

	always @(posedge clk or posedge rst) begin
		if (rst)
			branch <= 0;
		else
			branch <= v_branch;
	end


	// EX stage ALU control

	assign out_add = pipe_ce ? ~sub : add;
	assign out_ci = pipe_ce ?  (sub ^ (adcsbc & co)) : ci;

	always @(posedge clk or posedge rst) begin
		if (rst) begin
			add <= 1;
			ci <= 0;
		end
		else begin
			add <= v_add;
```

```
			ci <= v_ci;
		end
	end

	assign logicop		= ex_ir[5:4];
	assign sri		= ex_fn==`SRA && a15;

	// EX stage result multiplexer control

	assign out_sum_t =  pipe_ce ? ~(addsub || adcsbc) : sum_t;
	assign out_logic_t = pipe_ce ? ~(rrri&& (fn==`AND || fn==`OR || fn==`XOR
|| fn==`ANDN)) : logic_t;
	assign out_zeroext_t = pipe_ce ? ~(op==`LB) : zeroext_t;
	assign out_shr_t = pipe_ce ? ~(rrri && (fn==`SRL || fn==`SRA)) : shr_t;
	assign out_shl_t = pipe_ce ? ~(rrri && fn==`SLL) : shl_t;
	assign out_ret_t = pipe_ce ? ~(op==`JAL || op==`CALL) : ret_t;

	always @(posedge clk or posedge rst) begin
		if (rst) begin
			sum_t  <= 1;
			logic_t <= 1;
			zeroext_t <= 1;
			shr_t <= 1;
			shl_t <= 1;
			ret_t <= 1;
		end
		else begin
			sum_t  <= v_sum_t;
			logic_t <= v_logic_t;
			zeroext_t <= v_zeroext_t;
			shr_t <= v_shr_t;
			shl_t <= v_shl_t;
			ret_t <= v_ret_t;
		end
	end

endmodule


/* ctrl.v -- xr16 control unit synthesizable Verilog model
 *
 * Copyright (C) 1999, 2000, Gray Research LLC.  All rights reserved.
 * The contents of this file are subject to the XSOC License Agreement;
 * you may not use this file except in compliance with this Agreement.
 * See the LICENSE file.
 *
 * $Header: /dist/xsocv/ctrl.v 8     4/06/00 10:55a Jan $
 * $Log: /dist/xsocv/ctrl.v $
 *
 * 8     4/06/00 10:55a Jan
 * polish


// Instruction fields, ref.: The xr16 Specifications (doc/xspecs.pdf).

// Opcodes (IR[15:12])
`define ADD		4'h0
`define SUB		4'h1
`define	ADDI	4'h2
`define	RR		4'h3
`define RI		4'h4
`define LW		4'h5
```

```
`define LB           4'h6
`define SW           4'h8
`define SB           4'h9
`define JAL          4'hA
`define Bcond 4'hB
`define CALL  4'hC
`define IMM          4'hD

// Functions (IR[7:4])
`define AND          4'h0
`define OR           4'h1
`define XOR          4'h2
`define ANDN  4'h3
`define ADC          4'h4
`define SBC          4'h5
`define SRL          4'h6
`define SRA          4'h7
`define       SLL          4'h8

// Conditional branches (IR[11:8])
`define BR           4'h0
`define BRN          4'h1
`define       BEQ          4'h2
`define       BNE          4'h3
`define       BC           4'h4
`define       BNC          4'h5
`define       BV           4'h6
`define       BNV          4'h7
`define       BLT          4'h8
`define       BGE          4'h9
`define       BLE          4'hA
`define BGT          4'hB
`define       BLTU   4'hC
`define       BGEU   4'hD
`define       BLEU   4'hE
`define       BGTU   4'hF

`define INT          16'hAE01     // int ::= jal r14,10(r0)

module control(
       clk, rst, rdy, int_req, dma_req, zerodma, insn, a15, z, n, co, v,
       mem_ce, word_nxt, read_nxt, dbus_nxt, dma,
       rf_we, rna, rnb,
       fwd, imm, sextimm4, zextimm4, wordimm4, imm12, pipe_ce, b15_4_ce,
       add, ci, logicop, sri,
       sum_t, logic_t, shl_t, shr_t, zeroext_t, ret_t,
       branch, brdisp, selpc, zeropc, dmapc, pc_ce, ret_ce);

       parameter            W      = 16;  // data word width
       parameter            N      = W-1; // msb index
       parameter            IW     = 16;  // instruction word width
       parameter            IN     = 15;  // instruction word msb index

       // ports
       input                clk;          // global clock
       input                rst;          // global async reset
       input                rdy;          // current memory access is ready
       input                int_req;      // interrupt request
       input                dma_req;      // DMA request
       input                zerodma;      // zero DMA counter request
       input  [IN:0] insn;         // new instruction word
       input                a15;          // A operand msb
```

```
	input			z;			// zero result condition code
	input			n;			// negative result condition code
	input			co;			// carry-out result codition code
	input			v;			// oVerflow result condition code
	output			mem_ce;			// memory access clock enable
	output			word_nxt;	// next access is word wide
	output			read_nxt;	// next access is read
	output			dbus_nxt;	// next access uses on-chip data bus
	output			dma;		// current access is a DMA transfer
	output			rf_we;		// register file write enable
	output [3:0]	rna;		// register file port A register number
	output [3:0]	rnb;		// register file port B register number
	output			fwd;		// forward result bus into A operand
register
	output [11:0] imm;		// 12-bit immediate field
	output			sextimm4;	// sign-extend 4-bit immediate operand
	output			zextimm4;	// zero-extend 4-bit immediate operand
	output			wordimm4;	// word-offset 4-bit immediate operand
	output			imm12;		// 12-bit immediate operand
	output			pipe_ce;	// pipeline clock enable
	output			b15_4_ce;	// b[15:4] clock enable
	output			add;		// 1 => A + B; 0 => A - B
	output			ci;			// carry-in
	output [1:0]	logicop;	// logic unit opcode
	output			sri;		// shift right msb input
	output			sum_t;		// active low adder output enable
	output			logic_t;	// active low logic unit output enable
	output			shl_t;		// active low shift left output enable
	output			shr_t;		// active low shift right output enable
	output			zeroext_t;	// active low zero-extension output
enable
	output			ret_t;		// active low return address output
enable
	output			branch;			// branch taken
	output [7:0]	brdisp;			// 8-bit branch displacement
	output			selpc;		// address mux selects next PC
	output			zeropc;			// force next PC to 0
	output			dmapc;		// use DMA register in PC register file
	output			pc_ce;		// PC clock enable
	output			ret_ce;			// return address clock enable
	reg				add, ci, branch;
	reg				sum_t, logic_t, shl_t, shr_t, zeroext_t,
ret_t;

	// locals
	reg [IN:0]	if_ir, ir, ex_ir;	// instruction registers

	// IR fields
	wire [3:0]	op		= ir[15:12]; // opcode
	wire [3:0]	rd		= ir[11:8];		// destination register
	wire [3:0]	ra		= ir[7:4];		// source register A
	wire [3:0]	rb		= ir[3:0];		// source register B
	wire [3:0]	cond	= ir[11:8];		// branch condition
	wire [3:0]	fn		= ir[7:4];		// rr or ri function sub-
opcode
	wire [3:0]	ex_op	= ex_ir[15:12];		// EX stage opcode
	wire [3:0]	ex_rd	= ex_ir[11:8];		// EX stage destination register
	wire [3:0]	ex_fn	= ex_ir[7:4];// EX stage function sub-opcode
	assign		imm		= ir[11:0];		// 12-bit immediate field
	assign		brdisp = ex_ir[7:0];// 8-bit branch displacement

	// IF stage instruction decoding
```

```
	assign imm12 = op==`CALL || op==`IMM;
	assign sextimm4     = op==`ADDI || op==`RI;
	assign zextimm4     = op==`LB || op==`SB;
	assign wordimm4 = op==`LW || op==`SW || op==`JAL;
	wire addsub         = op==`ADD || op==`SUB || op==`ADDI;
	wire call           = op==`CALL;
	wire st                     = op==`SW || op==`SB;
	wire rrri           = op==`RR || op == `RI;
	wire adcsbc         = rrri && (fn==`ADC || fn==`SBC);
	wire dcintinh= op==`ADD || op==`SUB || op==`ADDI || op==`Bcond ||
	                            op==`CALL || op==`IMM || adcsbc;
	wire sub            = op==`SUB || (rrri && (fn==`SBC));

	// EX stage decoding
	wire ex_ldst = ex_op==`LW || ex_op==`LB || ex_op==`SW || ex_op==`SB;
	wire ex_lbsb = ex_op==`LB || ex_op==`SB;
	wire ex_results     = ex_op==`ADD || ex_op==`SUB || ex_op==`ADDI ||
	                            ex_op==`RR || ex_op==`RI ||
	                            ex_op==`LW || ex_op==`LB || ex_op==`JAL ||
ex_op==`CALL;
	wire ex_jump = (ex_op==`JAL || ex_op==`CALL);
	reg  ex_call;// EX stage call
	reg  ex_st;         // EX stage store

	// memory access FSM states -- each access is an instruction fetch,
	// unless DMA or load/store is pending
	reg  ifetch; // "current access is insn fetch"
	reg  dma;           // "current access is DMA"
	                          // "current access is load/store" ::=
~(ifetch|dma)
	// annul FSM states
	reg  sync_reset; // synchronous memory FSM reset
	reg  dc_annul;      // annul DC stage instruction
	reg  ex_annul;      // annul EX stage instruction

	// interrupt state
	reg  int_pend;      // interrupt is pending
	reg  dc_int; // DC stage instruction is int
	wire if_int; // IF stage 'interrupt in progress'

	// DMA state
	reg    dma_pend;    // DMA is pending
	reg  zero_pend;     // zero (reset) DMA is pending

	// memory access FSM transitions:
	// case (state)
	//   IF:  state_nxt = dma_pend ? DMA : ldst_pend ? LS : IF;
	//   DMA: state_nxt =                  ldst_pend ? LS : IF;
	//   LS:  state_nxt =                                   IF;
	// endcase
	wire ldst_pend      = ex_ldst & ~ex_annul;
	wire if_nxt         = ifetch&~dma_pend&~ldst_pend | dma&~ldst_pend |
	                            ~ifetch&~dma;
	wire dma_nxt  = ifetch&dma_pend;
	wire ldst_nxt = ifetch&~dma_pend&ldst_pend | dma&ldst_pend;

	wire jump           = ex_jump && ~ex_annul;
	wire exannul_nxt= sync_reset | branch | jump | dc_annul;

	// FSM output decodes
	assign mem_ce = rdy;
	assign pipe_ce      = rdy&if_nxt;
```

```
assign pc_ce  = rdy&(if_nxt|dma_nxt);
assign ret_ce = rdy&if_nxt&~dc_int;
assign word_nxt     = ~(ldst_nxt&ex_lbsb);
assign read_nxt     = ~(ldst_nxt&ex_st);
assign dbus_nxt     = ldst_nxt;
assign dmapc  = dma_nxt;
assign selpc  = if_nxt&~jump | dma_nxt;
assign zeropc = (zero_pend&dma_nxt) | sync_reset;
assign if_int = int_pend&~branch&~jump&~dcintinh;

// IR clocking
always @(posedge clk or posedge rst) begin
	if (rst) begin
		if_ir <= 0;
		ir <= 0;
		ex_ir <= 0;
	end
	else begin
		if (ifetch && rdy)
			if_ir <= insn;
		if (pipe_ce) begin
			ir <= (if_int) ? `INT : (ifetch) ? insn : if_ir;
			ex_ir <= ir;
		end
	end
end

// instruction decode clocking
always @(posedge clk or posedge rst) begin
	if (rst) begin
		ex_call <= 0;
		ex_st <= 0;
	end
	else if (pipe_ce) begin
		ex_call <= call;
		ex_st <= st;
	end
end

// memory access FSM clocking
always @(posedge clk or posedge rst) begin
	if (rst) begin
		ifetch <= 1;
		dma <= 0;
	end
	else if (rdy) begin
		ifetch <= if_nxt;
		dma <= dma_nxt;
	end
end

// annul states clocking
always @(posedge clk or posedge rst) begin
	if (rst)
		sync_reset <= 1;
	else if (rdy)
		sync_reset <= 0;
end
always @(posedge clk or posedge rst) begin
	if (rst) begin
		dc_annul <= 1;
		ex_annul <= 1;
```

```
		end
		else if (pipe_ce) begin
			dc_annul <= sync_reset | branch | jump;
			ex_annul <= exannul_nxt;
		end
	end

	// DMA request clocking
	always @(posedge clk or posedge rst) begin
		if (rst)
			dma_pend <= 0;
		else if (dma_req)
			dma_pend <= 1;
		else if (dma)
			dma_pend <= 0;
	end
	always @(posedge clk or posedge rst) begin
		if (rst)
			zero_pend <= 0;
		else if (zerodma)
			zero_pend <= 1;
		else if (dma)
			zero_pend <= 0;
	end

	// interrupt request clocking
	always @(posedge clk or posedge rst) begin
		if (rst)
			int_pend <= 0;
		else if (int_req)
			int_pend <= 1;
		else if (if_int && pipe_ce)
			int_pend <= 0;
	end
	always @(posedge clk or posedge rst) begin
		if (rst)
			dc_int <= 0;
		else if (pipe_ce)
			dc_int <= if_int;
	end

	// DC stage operand selection
	wire [3:0]	rsrc	= call ? 0 : rrri ? rd : ra;
	wire [3:0]	rdest	= ex_call ? 15 : ex_rd;
	assign		rna		= clk ? rsrc : rdest;
	assign		rnb		= clk ? (st ? rd : rb) : (ex_call ? 15 :
ex_rd);
	assign		rf_we	= ex_results & pipe_ce & ~ex_annul & (rdest != 0);
	assign		fwd		= (rsrc==rdest) & (rdest!=0) && ex_results &
~ex_annul;
	assign		b15_4_ce= pipe_ce && ~(ex_op==`IMM && ~ex_annul);

	// DC stage conditional branches
	reg taken;
	always @(posedge clk or posedge rst) begin
		if (rst)
			branch = 0;
		else if (pipe_ce) begin
			case (cond)
			`BR:	taken = 1;
			`BRN:	taken = 0;
			`BEQ:	taken = z;
```

```
                    `BNE:  taken = ~z;
                    `BC:   taken = co;
                    `BNC:  taken = ~co;
                    `BV:   taken = v;
                    `BNV:  taken = ~v;
                    `BLT:  taken = n^v;
                    `BGE:  taken = ~(n^v);
                    `BLE:  taken = (n^v)|z;
                    `BGT:  taken = ~((n^v)|z);
                    `BLTU: taken = ~z&~co;
                    `BGEU: taken = z|co;
                    `BLEU: taken = z|~co;
                    `BGTU: taken = ~z&co;
                    endcase
                    branch = op==`Bcond && taken && ~exannul_nxt;
             end
      end

      // EX stage ALU control
      always @(posedge clk or posedge rst) begin
             if (rst) begin
                    add <= 1;
                    ci <= 0;
             end
             else if (pipe_ce) begin
                    add <= ~sub;
                    ci <= sub ^ (adcsbc & co);
             end
      end
      assign logicop       = ex_ir[5:4];
      assign sri           = ex_fn==`SRA && a15;

      // EX stage result multiplexer control
      always @(posedge clk or posedge rst) begin
             if (rst) begin
                    sum_t  <= 1;
                    logic_t      <= 1;
                    zeroext_t <= 1;
                    shr_t  <= 1;
                    shl_t  <= 1;
                    ret_t  <= 1;
             end
             else if (pipe_ce) begin
                    sum_t  <= ~(addsub || adcsbc);
                    logic_t      <= ~(rrri&& (fn==`AND || fn==`OR || fn==`XOR
|| fn==`ANDN));
                    zeroext_t <= ~(op==`LB);
                    shr_t  <= ~(rrri && (fn==`SRL || fn==`SRA));
                    shl_t  <= ~(rrri && fn==`SLL);
                    ret_t  <= ~(op==`JAL || op==`CALL);
             end
      end

endmodule
 */
```

F. DATAPATH.V

```
`include     "voterids.v"
`timescale 1ns / 1ps
/* datapath.v -- xr16 datapath synthesizable Verilog model
 *
 * Copyright (C) 1999, 2000, Gray Research LLC.  All rights reserved.
 * The contents of this file are subject to the XSOC License Agreement;
 * you may not use this file except in compliance with this Agreement.
 * See the LICENSE file.
 *
 * $Header: /dist/xsocv/datapath.v 6     4/06/00 10:55a Jan $
 * $Log: /dist/xsocv/datapath.v $
 *
 * 6     4/06/00 10:55a Jan
 * polish
 */
 /* last modified by David Dwiggins, 24 August 2008.  Adapted to use
 *  TMR/ECC functionality for fault-tolerance.  */
 // original code by Jan Gray included at end of file

module datapath(clk, rst, rf_we, rna, rnb, rdest,fwd, imm,
              sextimm4, zextimm4, wordimm4, imm12, pipe_ce, b15_4_ce,
              add, ci, logicop, sri, sum_t, logic_t, shl_t, shr_t,
              zeroext_t, ret_t, ld_t, ud_t, udlt_t, branch, brdisp,
              selpc, zeropc, dmapc, pc_ce, ret_ce, in_a, in_b, in_dout,
              in_ret, inst_id, in_addr_nxt, res, errorbus,
              a15, z, n, co, v, addr_nxt,  areg, breg, memdta,
              out_a, out_b, out_dout, out_ret, drivehi, drivelo, my_res);

       parameter           W      = 16;  // word width
       parameter           N  = W-1;

       // ports
       input                     clk;          // global clock
       input                     rst;          // global async reset
       input  [2:0]        rf_we;        // register file write enable
       input [11:0] rna;          // register file port A register number
       input [11:0] rnb;          // register file port B register number
       input [11:0]   rdest;    // register file write port register number
       input                     fwd;          // forward result bus into A
operand reg
       input [11:0] imm;          // 12-bit immediate field
       input                     sextimm4;    // sign-extend 4-bit immediate
operand
       input                     zextimm4;    // zero-extend 4-bit immediate
operand
       input                     wordimm4;    // word-offset 4-bit immediate
operand
       input                     imm12;       // 12-bit immediate operand
       input  [2:0]        pipe_ce;     // pipeline clock enable
       input  [2:0]        b15_4_ce;    // b[15:4] clock enable
       input                     add;          // 1 => A + B; 0 => A - B
       input                     ci;                  // carry-in
       input [1:0]         logicop;     // logic unit opcode
       input                     sri;          // shift right msb input
       input                     sum_t;        // active low adder output enable
       input                     logic_t;      // active low logic unit output
enable
       input                     shl_t;        // active low shift left output
enable
```

```
	input			shr_t;		// active low shift right output
enable
	input			zeroext_t;	// active low zero-extension
output enable
	input			ret_t;		// active low return address
output enable
	input			ud_t;		// active low store data MSB
output enable
	input			ld_t;		// active low store data LSB
output enable
	input			udlt_t;			// active low store data
MSB->LSB output en
	input			branch;			// branch taken
	input [7:0]		brdisp;			// 8-bit branch displacement
	input			selpc;		// address mux selects next PC
	input			zeropc;			// force next PC to 0
	input  [2:0]		dmapc;		// use DMA register in PC register file
	input  [2:0]		pc_ce;		// PC clock enable
	input  [2:0]		ret_ce;			// return address clock enable
	input [47:0] in_a;
	input  [47:0] in_b;
	input [47:0] in_dout;
	input  [47:0] in_ret;
	input  [1:0]		inst_id;
	input  [47:0] in_addr_nxt;
	input  [47:0] res;		// on-chip data bus
	inout [15:0] errorbus;
	output		a15;		// A operand msb
	output		z;			// zero result condition code
	output		n;			// negative result condition code
	output		co;			// carry-out result codition code
	output		v;			// oVerflow result condition code
	output [N:0] addr_nxt;	// address of next memory access
	output [N:0]		areg;		// a register output, used for
debugging
	output [N:0]		breg;		// b register output, used for
debugging
	output [N:0]		memdta;		// B register output for memory writes
	output [15:0] out_a;		// a operand out (for voting)
	output [15:0] out_b;	// b operand out (for voting)
	output [15:0] out_dout;	// dout out (for voting)
	output [15:0] out_ret;		// ret out (for voting)
	output		drivehi;		// tri-state driver enable, high
byte
	output		drivelo;		// tri-state driver enable, low
byte
	output [15:0] my_res;		// data out

	wor	[15:0] errorbus;	// global error bus
	tri	[15:0] out_res;		// result out (for voting)



//	assign outerrorbus = errorbus;

	// voted buses

	wire		v_rf_we;		// register file write enable
	wire [3:0]	v_rna;		// register file port A register number
	wire [3:0]	v_rnb;		// register file port B register number
	wire [3:0]  v_rdest;	// register file write port register number
	wire		v_pipe_ce;	// pipeline clock enable
```

```
	wire			v_b15_4_ce;	// b[15:4] clock enable
	wire			v_dmapc;		// use DMA register in PC
register file
	wire			v_pc_ce;		// PC clock enable
	wire			v_ret_ce;		// return address clock enable

	wire	[15:0]	v_res;		// voted result bus
	wire	[15:0]	v_a;
	wire	[15:0]	v_b;
	wire	[15:0]	v_dout;
	wire	[15:0]	v_ret;
	wire	[15:0]	v_addr_nxt;
	wire	[47:0]	in_addr_nxt;

	wire		[15:0]	areg, breg; // outputs of the register file data
ports
	wire		[15:0]	pc;			// current PC
	wire		[6:0]	brdispext;


	reg		[N:0]	a, b;		// operand registers
	reg		[N:0]	dout;		// store data output register
	reg		[N:0]	ret;		// return address (DC stage PC)

	// non-clocked combinatorial verilog "registers"

	reg		[N:0]	sum;		// adder output
	reg		z, n, co, v;
	reg		[N:0]	addr_nxt;
	reg		[N:0]	pcnext;			// next PC	// REVIEW
	reg		[N:0]	dlogic;			// logic unit output

	assign brdispext = {brdisp[7],brdisp[7],brdisp[7],brdisp[7],

brdisp[7],brdisp[7],brdisp[7]};


	voter1nsyn #(1)		rf_we_v (.din(rf_we), .dout(v_rf_we),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`DP_RF_WE_V}));
	voter1nsyn #(4)		rna_v (.din(rna), .dout(v_rna), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID({`DP_RNA_V}));
	voter1nsyn #(4)		rnb_v (.din(rnb), .dout(v_rnb), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID({`DP_RNB_V}));
	voter1nsyn #(4)		rdest_v (.din(rdest), .dout(v_rdest),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`DP_RDEST_V}));
	voter1nsyn #(1)		dmapc_v (.din(dmapc), .dout(v_dmapc),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`DP_DMAPC_V}));
	voter1nsyn #(1)		pipe_ce_v (.din(pipe_ce), .dout(v_pipe_ce),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`DP_PIPE_CE_V}));
	voter1nsyn #(1)		b15_4_ce_v (.din(b15_4_ce), .dout(v_b15_4_ce),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`DP_B15_4_CE_V}));

	voter1nsyn #(16)	addr_nxt_v (.din(in_addr_nxt), .dout(v_addr_nxt),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`DP_ADDR_NXT_V}));
	voter1nsyn #(1)		pc_ce_v (.din(pc_ce), .dout(v_pc_ce),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`DP_PC_CE_V}));
	voter1nsyn #(1)		ret_ce_v (.din(ret_ce), .dout(v_ret_ce),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`DP_RET_CE_V}));
	voter1nsyn #(16)	res_v (.din(res), .dout(v_res), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID({`DP_RES_V}));
	voter1nsyn #(16)	a_v (.din(in_a), .dout(v_a), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID({`DP_A_V}));
```

```
	voter1nsyn #(16)	b_v (.din(in_b), .dout(v_b), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID({`DP_B_V}));
	voter1nsyn #(16)	dout_v (.din(in_dout), .dout(v_dout),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`DP_DOUT_V}));
	voter1nsyn #(16)	ret_v (.din(in_ret), .dout(v_ret),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`DP_RET_V}));

	register16x16 abregs (.clk(clk), .rst(rst), .we(v_rf_we), .addra(v_rna),
.addrb(v_rnb),
		.addrw(v_rdest), .din(v_res), .adout(areg), .bdout(breg));

	pcfile pcs (.clk(clk), .rst(rst), .we(v_pc_ce), .addr(v_dmapc),
.din(v_addr_nxt), .adout(pc));

	assign memdta=v_pipe_ce ? breg : dout;

	// operand registers

	always @(posedge clk or posedge rst) begin
		if (rst) begin
			a <= 0;
			b <= 0;
			dout <= 0;
		end
		else begin
			if (v_pipe_ce) begin
				a <= v_a;
				b[3:0] <= v_b[3:0];
				dout <= v_dout;
			end
			if (v_b15_4_ce) begin
				b[15:4] <= v_b[15:4];
			end
		end
	end

	assign out_dout = breg;
	assign out_a = fwd ? res : areg;

	assign out_b[3:0] = (sextimm4 || zextimm4) ? imm[3:0] : ( wordimm4 ?
({imm[3:1],1'b0}) : (imm12 ? 4'b0 : breg[3:0]));
	assign out_b[15:4] = sextimm4 ? ({12{imm[3]}}) : (zextimm4 ? 12'b0 :
(wordimm4 ? ({11'b0,imm[0]}) : (imm12 ? imm[11:0] : breg[15:4])));

	//ISE understands the following but ModelSim does not, ugly assign code
above followed
	//always_comb
	//			if (v_sextimm4 || v_zextimm4)
	//				out_b[3:0] <= imm[3:0];
	//			else if (v_wordimm4)
	//				out_b[3:0] <= {imm[3:1],1'b0};
	//			else if (v_imm12)
	//				out_b[3:0] <= 0;
	//			else
	//				out_b[3:0] <= breg[3:0];
	//always_comb
	//			if (v_sextimm4)
	//				out_b[15:4] <= {12{imm[3]}};
	//			else if (v_zextimm4)
	//				out_b[15:4] <= 12'b0;
	//			else if (v_wordimm4)
	//				out_b[15:4] <= {11'b0,imm[0]};
```

```
	//			else if (v_imm12)
	//				out_b[15:4] <= imm[11:0];
	//			else
	//				out_b[15:4] <= breg[15:4];

	// ALU
	assign a15 = a[15];
	reg ign;
	reg co0;
	always @(a or b or add or ci) begin
		if (add) begin
			{co0,sum,ign} = {a,ci} + {b,1'b1};
			co = co0;
		end
		else begin
			{co0,sum,ign} = {a,ci} - {b,1'b1};
			co = ~co0;
		end
		z = sum == 0;
		n = sum[15];
		// sum[15] = a[15] ^ b[15] ^ c15 <===> c15 = sum[15] ^ a[15] ^
b[15]
		v = co0 ^ sum[15] ^ a[15] ^ b[15];
	end
	always @(a or b or logicop) begin
		case (logicop)
		0: dlogic <= a&b;
		1: dlogic <= a|b;
		2: dlogic <= a^b;
		3: dlogic <= a&~b;
		endcase
	end

	// address/PC unit
	always @(branch or pc or brdispext or brdisp or dmapc) begin
		if (branch & ~dmapc)
			pcnext <= pc + {brdispext[6:0],brdisp[7:0],1'b0};
		else
			pcnext <= pc + 2;
	end
	always @(sum or pcnext or selpc or zeropc) begin
		if (zeropc)
			addr_nxt <= 0;
		else if (selpc)
			addr_nxt <= pcnext;
		else
			addr_nxt <= sum;
	end

	assign out_ret = pc;

	always @(posedge clk or posedge rst) begin
		if (rst)
			ret <= 0;
		else if (v_ret_ce)
			ret <= v_ret;
	end

	// result multiplexer
	assign out_res		= sum_t	? 16'bz : sum;
	assign out_res		= logic_t	? 16'bz : dlogic;
	assign out_res		= shl_t	? 16'bz : { a[14:0],1'b0 };
```

```
	assign out_res             = shr_t     ? 16'bz : { sri, a[15:1] };
	assign out_res             = ret_t     ? 16'bz : ret;
	assign out_res[7:0]  = ld_t      ? 8'bz  : dout[7:0];
	assign out_res[15:8] = ud_t      ? 8'bz  : dout[15:8];
	assign out_res[7:0]  = udlt_t    ? 8'bz  : dout[15:8];
	assign out_res[15:8] = zeroext_t ? 8'bz  : 0;


	// due to partitioning requirements, moving the tri-state data bus
drivers outside the module
	assign my_res = out_res;
	assign drivehi = ((~sum_t) | (~logic_t) | (~shl_t) | (~shr_t) | (~ret_t)
| (~ud_t) | (~zeroext_t));
	assign drivelo = ((~sum_t) | (~logic_t) | (~shl_t) | (~shr_t) | (~ret_t)
| (~ld_t) | (~udlt_t));

endmodule

// original code by Jan Gray below

/* datapath.v -- xr16 datapath synthesizable Verilog model
 *
 * Copyright (C) 1999, 2000, Gray Research LLC.  All rights reserved.
 * The contents of this file are subject to the XSOC License Agreement;
 * you may not use this file except in compliance with this Agreement.
 * See the LICENSE file.
 *
 * $Header: /dist/xsocv/datapath.v 6     4/06/00 10:55a Jan $
 * $Log: /dist/xsocv/datapath.v $
 *
 * 6     4/06/00 10:55a Jan
 * polish


module datapath(
		clk, rst,
		rf_we, rna, rnb,
		fwd, imm, sextimm4, zextimm4, wordimm4, imm12, pipe_ce, b15_4_ce,
		add, ci, logicop, sri,
		sum_t, logic_t, shl_t, shr_t, zeroext_t, ret_t, ld_t, ud_t,
udlt_t,
		branch, brdisp, selpc, zeropc, dmapc, pc_ce, ret_ce,
		a15, z, n, co, v, addr_nxt, res);

	parameter          W      = 16;  // word width
	parameter          N      = W-1; // msb index

	// ports
	input              clk;          // global clock
	input              rst;          // global async reset
	input              rf_we;        // register file write enable
	input [3:0]        rna;          // register file port A register number
	input [3:0]        rnb;          // register file port B register number
	input              fwd;          // forward result bus into A operand reg
	input [11:0]  imm;          // 12-bit immediate field
	input              sextimm4;     // sign-extend 4-bit immediate operand
	input              zextimm4;     // zero-extend 4-bit immediate operand
	input              wordimm4;     // word-offset 4-bit immediate operand
	input              imm12;        // 12-bit immediate operand
	input              pipe_ce;      // pipeline clock enable
	input              b15_4_ce;     // b[15:4] clock enable
	input              add;          // 1 => A + B; 0 => A - B
```

```
	input			ci;			// carry-in
	input [1:0]		logicop;	// logic unit opcode
	input			sri;		// shift right msb input
	input			sum_t;		// active low adder output enable
	input			logic_t;	// active low logic unit output enable
	input			shl_t;		// active low shift left output enable
	input			shr_t;		// active low shift right output enable
	input			zeroext_t;	// active low zero-extension output
enable
	input			ret_t;		// active low return address output
enable
	input			ud_t;		// active low store data MSB output
enable
	input			ld_t;		// active low store data LSB output
enable
	input			udlt_t;			// active low store data MSB->LSB
output en
	input			branch;			// branch taken
	input [7:0]		brdisp;			// 8-bit branch displacement
	input			selpc;		// address mux selects next PC
	input			zeropc;			// force next PC to 0
	input			dmapc;		// use DMA register in PC register file
	input			pc_ce;		// PC clock enable
	input			ret_ce;			// return address clock enable
	output			a15;		// A operand msb
	output			z;			// zero result condition code
	output			n;			// negative result condition code
	output			co;			// carry-out result codition code
	output			v;			// oVerflow result condition code
	output [N:0]	addr_nxt;	// address of next memory access
	inout  [N:0]	res;		// on-chip data bus
	tri		[N:0]	res;

	reg				z, n, co, v;
	reg		[N:0]	addr_nxt;

	// locals
	wire [N:0]	areg_nxt, breg_nxt; // reg file read port RAM output buses
	reg	  [N:0] areg, breg;	// register file read ports
	reg  [N:0]	a, b;		// operand registers
	reg  [N:0]	dout;		// store data output register
	reg  [N:0]	sum;		// adder output
	reg  [N:0]	logic;		// logic unit output
	wire [N:0]	pc;			// current PC
	reg  [N:0]	pcnext;		// next PC	// REVIEW
	reg  [N:0]	ret;		// return address (DC stage PC)
	wire [6:0]	brdispext = {brdisp[7],brdisp[7],brdisp[7],brdisp[7],

brdisp[7],brdisp[7],brdisp[7]};

	// submodules
	ram16x16s aregs(.wclk(clk), .addr(rna), .we(rf_we), .d(res),
.o(areg_nxt));
	ram16x16s bregs(.wclk(clk), .addr(rnb), .we(rf_we), .d(res),
.o(breg_nxt));
	ram16x16s pcs(.wclk(clk), .addr({3'b0,dmapc}), .we(pc_ce), .d(addr_nxt),
				.o(pc));

	// register file output registers
	always @(negedge clk or posedge rst) begin
		if (rst) begin
			areg <= 0;
```

```
                breg <= 0;
            end
            else begin
                areg <= areg_nxt;
                breg <= breg_nxt;
            end
        end

        // operand registers
        always @(posedge clk or posedge rst) begin
            if (rst) begin
                a <= 0;
                b <= 0;
                dout <= 0;
            end
            else begin
                if (pipe_ce) begin
                    a <= fwd ? res : areg;
                    if (sextimm4 || zextimm4)
                        b[3:0] <= imm[3:0];
                    else if (wordimm4)
                        b[3:0] <= {imm[3:1],1'b0};
                    else if (imm12)
                        b[3:0] <= 0;
                    else
                        b[3:0] <= breg[3:0];
                    dout <= breg;
                end
                if (b15_4_ce) begin
                    if (sextimm4)
                        b[15:4] <= {12{imm[3]}};
                    else if (zextimm4)
                        b[15:4] <= 12'b0;
                    else if (wordimm4)
                        b[15:4] <= {11'b0,imm[0]};
                    else if (imm12)
                        b[15:4] <= imm[11:0];
                    else
                        b[15:4] <= breg[15:4];
                end
            end
        end

        // ALU
        assign a15 = a[15];
        reg ign;
        reg co0;
        always @(a or b or add or ci) begin
            if (add) begin
                {co0,sum,ign} = {a,ci} + {b,1'b1};
                co = co0;
            end
            else begin
                {co0,sum,ign} = {a,ci} - {b,1'b1};
                co = ~co0;
            end
            z = sum == 0;
            n = sum[15];
            // sum[15] = a[15] ^ b[15] ^ c15 <===> c15 = sum[15] ^ a[15] ^
b[15]
            v = co0 ^ sum[15] ^ a[15] ^ b[15];
        end
```

```
	always @(a or b or logicop) begin
		case (logicop)
		0: logic <= a&b;
		1: logic <= a|b;
		2: logic <= a^b;
		3: logic <= a&~b;
		endcase
	end

	// address/PC unit
	always @(branch or pc or brdispext or brdisp or dmapc) begin
		if (branch & ~dmapc)
			pcnext <= pc + {brdispext[6:0],brdisp[7:0],1'b0};
		else
			pcnext <= pc + 2;
	end
	always @(sum or pcnext or selpc or zeropc) begin
		if (zeropc)
			addr_nxt <= 0;
		else if (selpc)
			addr_nxt <= pcnext;
		else
			addr_nxt <= sum;
	end
	always @(posedge clk or posedge rst) begin
		if (rst)
			ret <= 0;
		else if (ret_ce)
			ret <= pc;
	end

	// result multiplexer
	assign res          = sum_t     ? 16'bz : sum;
	assign res          = logic_t   ? 16'bz : logic;
	assign res          = shl_t     ? 16'bz : { a[14:0],1'b0 };
	assign res          = shr_t     ? 16'bz : { sri, a[15:1] };
	assign res          = ret_t     ? 16'bz : ret;
	assign res[7:0]     = ld_t      ? 8'bz  : dout[7:0];
	assign res[15:8] = ud_t      ? 8'bz  : dout[15:8];
	assign res[7:0]     = udlt_t    ? 8'bz  : dout[15:8];
	assign res[15:8] = zeroext_t ? 8'bz  : 0;

endmodule


*/
```

G. ERRORFIFO.V

```
`include "voterids.v"
`timescale 1ns / 1ps
`define	ERROR_DATA		5'h0
`define	ERROR_HAS_DATA		5'h2

module errorrept(clk, rst, ctrl, int_req, sel, d, errorbus);
	input				clk;		// global clock
	input				rst;		// global async reset
	input  [47:0]		ctrl;		// abstract control bus
	output [2:0]		int_req;
	input  [2:0]			sel;		// peripheral select
```

```
	inout	[47:0]		d;			// LSB of on-chip data bus
	inout	[15:0]		errorbus;

	tri	[47:0]		d;
	wire	[2:0]			int_req;

	wor	[15:0]		errorbus;
	wire [14:0]			addr;
	wire	[2:0]			ud_t, ld_t, ud_ce, ld_ce;
	wire	[47:0]			dout;
	wire	[2:0]			er_add;
	wire	[2:0]			er_remove;
	wire	[15:0]		queue_out;
	wire				full;
	wire				nearlyfull;

	ctrl_dec_syn dec0(.clk(clk), .rst(rst), .ctrl(ctrl), .sel(sel),
.ud_t(ud_t[0]),
				.ld_t(ld_t[0]), .ud_ce(ud_ce[0]), .ld_ce(ld_ce[0]),
.addr(addr[4:0]),
				.errorbus(errorbus), .id(`ER_CTRL_DEC));

	ctrl_dec_syn dec1(.clk(clk), .rst(rst), .ctrl(ctrl), .sel(sel),
.ud_t(ud_t[1]),
				.ld_t(ld_t[1]), .ud_ce(ud_ce[1]), .ld_ce(ld_ce[1]),
.addr(addr[9:5]),
				.errorbus(errorbus), .id(`ER_CTRL_DEC));


	ctrl_dec_syn dec2(.clk(clk), .rst(rst), .ctrl(ctrl), .sel(sel),
.ud_t(ud_t[2]),
				.ld_t(ld_t[2]), .ud_ce(ud_ce[2]), .ld_ce(ld_ce[2]),
.addr(addr[14:10]),
				.errorbus(errorbus), .id(`ER_CTRL_DEC));

	errorrept_mod err_rpt0(.clk(clk), .rst(rst), .addr(addr),
.errorbus(errorbus),
				.ld_t(ld_t), .dout(dout[15:0]), .er_add(er_add[0]),
.er_remove(er_remove[0]),
				.queue_empty(empty), .queue_out(queue_out),
				.int_req(int_req[0]));

	errorrept_mod err_rpt1(.clk(clk), .rst(rst), .addr(addr),
.errorbus(errorbus),
				.ld_t(ld_t), .dout(dout[31:16]), .er_add(er_add[1]),
.er_remove(er_remove[1]),
				.queue_empty(empty), .queue_out(queue_out),
				.int_req(int_req[1]));

	errorrept_mod err_rpt2(.clk(clk), .rst(rst), .addr(addr),
.errorbus(errorbus),
				.ld_t(ld_t), .dout(dout[47:32]), .er_add(er_add[2]),
.er_remove(er_remove[2]),
				.queue_empty(empty), .queue_out(queue_out),
				.int_req(int_req[2]));

	queue_16x8 errorqueue (.din(errorbus), .clk(clk), .rst(rst),
.add(er_add), .remove(er_remove),
			.dout(queue_out), .empty(empty), .full(full),
.nearlyfull(nearlyfull), .errorbus(errorbus));

	assign d[7:0] = ld_t[0] ? 8'bz : dout[7:0];
```

```
	assign d[15:8] = ud_t[0] ? 8'bz : dout[15:8];
	assign d[23:16] = ld_t[1] ? 8'bz : dout[23:16];
	assign d[31:24] = ud_t[1] ? 8'bz : dout[31:24];
	assign d[39:32] = ld_t[2] ? 8'bz : dout[39:32];
	assign d[47:40] = ud_t[2] ? 8'bz : dout[47:40];

endmodule

module errorrept_mod(clk, rst, errorbus, addr, ld_t, dout,
					er_add, er_remove, queue_empty,
queue_out,
					int_req);
	input			clk;		// global clock
	input			rst;		// global async reset
	inout	[15:0]	errorbus;
	input [14:0]	addr;
	input [2:0]		ld_t;
	output [15:0]		dout;
	output			er_add;
	output			er_remove;
	input				queue_empty;
	input	[15:0]	queue_out;
	output			int_req;


	wor	[15:0]	errorbus;
	wire [14:0]		addr;
	wire	[2:0]		ld_t;
	reg	[15:0]		dout;
	wire [15:0]		queue_out;
	wire				v_ld_t;
	wire				empty;
	wire				full;
	wire				nearlyfull;
	wire [4:0]		v_addr;
	wire				remove;

	assign er_add = |errorbus;
	assign er_remove = ~v_ld_t & (v_addr == `ERROR_DATA);  // removing data
on read of error data address
	assign int_req = ~empty;

	voter1nsyn #(1)	ld_t_v (.din(ld_t), .dout(v_ld_t),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID(`ER_MO_LD_T_V));
	voter1nsyn #(5)	addr_v (.din(addr), .dout(v_addr),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID(`ER_MO_ADDR_V));

	always @(*) begin
		case (v_addr)
			`ERROR_DATA:
				dout <= queue_out;
			`ERROR_HAS_DATA:
				dout <= {15'b0,~empty};
			default:
				dout <= 16'b0;
		endcase
	end

endmodule
```

H. IOTEMPLATE.V

```
`timescale 1ns / 1ps
`define      MY_MODULE_CTRL_DEC_SYN_ID        10'd192
`define      MY_MODULE_DATA_BUS_ID            10'd193
`define      MY_MODULE_LD_CE_ID               10'd194
`define      MY_MODULE_ADDR_ID                10'd195
`define      MY_IO_ADDR                       5'd0

module my_io_module(clk, rst, addr, din, dout,
                                 errorbus, ld_t, ud_t,
                                 ld_ce, ud_ce);

       input                     clk;
       input                     rst;
       input  [14:0] addr;
       input  [15:0] din;
       output [15:0] dout;
       inout  [15:0] errorbus;
       input   [2:0] ld_t;
       input   [2:0] ud_t;
       input   [2:0] ld_ce;
       input   [2:0] ud_ce;

       wire                      clk;
       wire                      rst;
       wire    [4:0] addr;
       wire    [15:0]      din;
       wire   [15:0] dout;
       wor                       errorbus;
       wire                      ld_t;
       wire                      ud_t;
       wire                      ld_ce;
       wire                      ud_ce;

       reg [15:0] my_sample_reg;
       reg    [15:0] dout;
       wire   [15:0] v_d;
       wire                v_ld_ce;
       wire                v_addr;

       voter1nsyn  #(16) d_v (.din(d), .dout(v_d), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID(`MY_MODULE_DATA_BUS_ID));
       voter1nsyn  #(1) ld_ce_v (.din(ld_ce), .dout(v_ld_ce),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID(`MY_MODULE_LD_CE_ID));
       voter1nsyn  #(1) addr_v (.din(addr), .dout(v_addr), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID(`MY_MODULE_ADDR_ID));

       always @(posedge clk or posedge rst) begin
       if (rst) begin
              my_sample_reg <= 0;
       end
       else if (v_ld_ce) begin
              if(v_addr == `MY_IO_ADDR) begin
                     my_sample_reg <= v_d;  // load data into the register
              end
       end

       end

       always @(v_addr or my_sample_reg) begin
```

```
            if(v_addr == `MY_IO_ADDR) begin
                  dout <= my_sample_reg;     // output the register contents
if the correct address value
            end
            else begin
                  dout <= 16'b0;
            end
      end

endmodule


module iotemplate(clk, rst, ctrl, ctrl0, int_req, sel, d, errorbus);
   input                  clk;
   input                  rst;
   input [47:0]     ctrl;
      inout  [2:0]        ctrl0;
      output [2:0] int_req;
   input [2:0]      sel;
   inout [47:0]     d;
      inout [15:0] errorbus;

      wire                       clk;          // Clock
      wire                       rst;          // Reset
      wire   [47:0] ctrl;        //     16-bit memory control bus
      wire   [2:0]        sel;          // 1-bit peripheral select # (IO Base
Select)
                                                            // I/O 0 is at base
address FF00, subsequent IDs
                                                            //     are each 20h
higher
      tri    [47:0] d;                  // 16-bit tri-state data bus
      wor    [15:0]       errorbus;//   16-bit Global Error Bus, wired-OR
      wand   [2:0]        ctrl0; // drive low to insert IO wait states
      wire   [2:0]        int_req; // drive high to enable ISR

      wire   [14:0] addr;        // 5-bit Peripheral Address Space
      wire   [2:0]        ud_t;         // Active-low upper byte data bus tri-
state drive enable
      wire   [2:0]        ld_t;         // Active-low lower byte data bus tri-
state drive enable
      wire   [2:0]        ud_ce; // Active-high upper byte valid data clock
enable
      wire   [2:0]        ld_ce; // Active-high lower byte valid data clock
enable
      wire   [47:0] dout;        // 16-bit data bus


      // declare the XSOC IO Control modules
      //     each module should be partitioned to avoid
      // unwanted optimization (and removal of the 2 "extra" copies)
      ctrl_dec_syn dec0(.clk(clk), .rst(rst), .ctrl(ctrl), .sel(sel),
.ud_t(ud_t[0]),
                            .ld_t(ld_t[0]), .ud_ce(ud_ce[0]), .ld_ce(ld_ce[0]),
.addr(addr[4:0]),
                            .id(`MY_MODULE_CTRL_DEC_SYN_ID),
.errorbus(errorbus));
      ctrl_dec_syn dec1(.clk(clk), .rst(rst), .ctrl(ctrl), .sel(sel),
.ud_t(ud_t[1]),
                            .ld_t(ld_t[1]), .ud_ce(ud_ce[1]), .ld_ce(ld_ce[1]),
.addr(addr[9:5]),
```

```
                            .id(`MY_MODULE_CTRL_DEC_SYN_ID),
.errorbus(errorbus));
       ctrl_dec_syn  dec2(.clk(clk), .rst(rst), .ctrl(ctrl), .sel(sel),
.ud_t(ud_t[2]),
                            .ld_t(ld_t[2]), .ud_ce(ud_ce[2]), .ld_ce(ld_ce[2]),
.addr(addr[14:10]),
                            .id(`MY_MODULE_CTRL_DEC_SYN_ID),
.errorbus(errorbus));

       // instantiate your IO module
       // also partition each module
       // note that ld_t and ud_t are used to indicate a read from IO
       // and ld_ce and ud_ce are used to indicate a write
       // data in and data out are separate because partitions do not allow tri-
state parameters
       // din must be declared as an input
       // generate....for loops aren't supported in partitions so instantiate
each explicitly
       my_io_module my_io_module_0(
                                  .clk(clk), .rst(rst), .addr(addr),
.din(d[15:0]), .dout(dout[15:0]),
                                  .errorbus(errorbus),.ld_t(ld_t), .ud_t(ud_t),
                                  .ld_ce(ld_ce), .ud_ce(ud_ce));

       my_io_module my_io_module_1(
                                  .clk(clk), .rst(rst), .addr(addr),
.din(d[31:16]), .dout(dout[31:16]),
                                  .errorbus(errorbus),.ld_t(ld_t), .ud_t(ud_t),
                                  .ld_ce(ld_ce), .ud_ce(ud_ce));

       my_io_module my_io_module_2(
                                  .clk(clk), .rst(rst), .addr(addr),
.din(d[47:32]), .dout(dout[47:32]),
                                  .errorbus(errorbus),.ld_t(ld_t), .ud_t(ud_t),
                                  .ld_ce(ld_ce), .ud_ce(ud_ce));


       // Data bus writes are accomplished here since partitions
       // do not allow a tri-state bus interface

       assign d[7:0]              = ld_t[0] ? 8'bz : dout[7:0];
       assign d[15:8]             = ud_t[0] ? 8'bz : dout[15:8];
       assign d[23:16]     = ld_t[1] ? 8'bz : dout[23:16];
       assign d[31:24]     = ud_t[1] ? 8'bz : dout[31:24];
       assign d[39:32]     = ld_t[2] ? 8'bz : dout[39:32];
       assign d[47:40]     = ud_t[2] ? 8'bz : dout[47:40];

endmodule
```

I. MEMCTRL.V

```
`timescale 1ns / 1ps
`include "voterids.v"
/* memctrl.v -- XSOC memory/on-chip bus controller synthesizable Verilog model
 *
 * Copyright (C) 1999, 2000, Gray Research LLC.  All rights reserved.
 * The contents of this file are subject to the XSOC License Agreement;
 * you may not use this file except in compliance with this Agreement.
 * See the LICENSE file.
```

```
 *
 * $Header: /dist/xsocv/memctrl.v 6     4/06/00 10:55a Jan $
 * $Log: /dist/xsocv/memctrl.v $
 *
 * 6     4/06/00 10:55a Jan
 * polish
 */
 /* modified 08/01/2008 by David Dwiggins for use with CFTP */
 /* TMR/ECC support.  Due to extensive modification */
 /* original version by Jan Gray commented out at the end of file */


module memctrl(
       clk, rst,
       mem_ce, word_nxt, read_nxt, dbus_nxt, dma, addr_nxt,
       dma_req_in, zero_req_in,ctrl0,
       rdy, ud_t, ld_t, udlt_t, dma_req, zerodma,
       uxd_t, lxd_t, store_ub, store_lb, read_ub, write_en,
       sel, ctrl, dma_ack, errorbus);

       parameter           W      = 16;  // word width
       parameter           N      = W-1; // msb index
       parameter           XAN    = 16;  // external address msb index

       input               clk;          // global clock
       input               rst;          // global async reset
       input  [2:0]  mem_ce;             // memory access clock enable
       input  [2:0]  word_nxt;     // next access is word wide
       input  [2:0]  read_nxt;     // next access is read
       input  [2:0]        dbus_nxt;     // next access uses on-chip data bus
       input  [2:0]        dma;          // current access is a DMA transfer
       input  [47:0] addr_nxt;     // address of next memory access
       input  [2:0]        dma_req_in;
       input  [2:0]        zero_req_in;
       input               ctrl0; // control 0 driven low => add wait states
       output              rdy;          // current memory access is ready
       output              ud_t;         // active low store data MSB output
enable
       output              ld_t;         // active low store data LSB output
enable
       output              udlt_t;             // active low store data MSB->LSB
output enable
       output              dma_req;      // DMA request
       output              zerodma;      // zero DMA counter request
       output              store_ub;           // ram store lsb into upper byte
(byte op)
       output              store_lb;           // ram store lsb into lsb (byte
op)
       output              read_ub;            // ram read upper byte into data
bus lsb
       output              write_en;           // ram write enable
       output              uxd_t; // RAM MSB data out enable
       output              lxd_t;    // RAM LSB data out enable
       output [7:0]  sel;          // on-chip peripheral select
       output [15:0] ctrl;         // abstract control bus
       output              dma_ack;      // DMA ackwnowledge
       inout  [15:0] errorbus;


       wor    [15:0] errorbus;
       reg                        uxd_t, lxd_t;
```

```
// locals
wire                selio;



wire          v_mem_ce;
wire          v_word_nxt;
wire          v_read_nxt;
wire          v_dbus_nxt;
wire          v_dma;
wire          [15:0] v_addr_nxt;
wire          v_dma_req_in;
wire          v_zero_req_in;


wire          ud_t_nxt;
wire          ld_t_nxt;
wire          ud_ce_nxt;
wire          ld_ce_nxt;
wire          ramrd_nxt;
wire          ramwr_nxt;

reg [4:0]     addr;  // current access address
reg                 word;  // current access is word wide
reg                 read;  // current access is read
reg                 io;           // current access is to I/O space
reg                 ramrd; // RAM read access
reg                 ramwr; // RAM write access
reg                 w1;           // byte write read cycle
reg                 w2;           // byte write write cycle
reg                 read_ub;           // read upper byte on next cycle

assign              selio  = v_addr_nxt[15:8] == 8'hFF;
assign        sel[0] = v_addr_nxt[7:5] == 0;
assign        sel[1] = v_addr_nxt[7:5] == 1;
assign        sel[2] = v_addr_nxt[7:5] == 2;
assign        sel[3] = v_addr_nxt[7:5] == 3;
assign        sel[4] = v_addr_nxt[7:5] == 4;
assign        sel[5] = v_addr_nxt[7:5] == 5;
assign        sel[6] = v_addr_nxt[7:5] == 6;
assign        sel[7] = v_addr_nxt[7:5] == 7;

assign        ud_t_nxt      = ~(v_word_nxt & v_read_nxt);
assign        ld_t_nxt      = ~v_read_nxt;
assign        ud_ce_nxt     = v_word_nxt & ~v_read_nxt;
assign        ld_ce_nxt     = ~v_read_nxt;
assign        ramrd_nxt = v_read_nxt & ~selio & v_mem_ce;
assign        ramwr_nxt = ~v_read_nxt & ~selio & v_mem_ce;



      voter1nsyn   #(1)   mem_ce_v (.din(mem_ce), .dout(v_mem_ce),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`MC_MEM_CE_V}));
      voter1nsyn   #(1)  word_nxt_v (.din(word_nxt), .dout(v_word_nxt),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`MC_WORD_NXT_V}));
      voter1nsyn  #(1)    read_nxt_v (.din(read_nxt), .dout(v_read_nxt),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`MC_READ_NXT_V}));
      voter1nsyn  #(1)  dbus_nxt_v (.din(dbus_nxt), .dout(v_dbus_nxt),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`MC_DBUS_NXT_V}));
      voter1nsyn  #(1)  dma_v (.din(dma), .dout(v_dma), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID({`MC_DMA_V}));
```

```
	voter1nsyn  #(16) addr_nxt_v (.din(addr_nxt), .dout(v_addr_nxt),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`MC_ADDR_NXT_V}));
	voter1nsyn  #(1)    dma_req_in_v (.din(dma_req_in), .dout(v_dma_req_in),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID({`MC_DMA_REQ_IN_V}));
	voter1nsyn  #(1)    zero_req_in_v (.din(zero_req_in),
.dout(v_zero_req_in), .errorbus(errorbus), .clk(clk), .rst(rst),
.ID({`MC_ZERO_REQ_IN_V}));

	assign       rdy          = (io&ctrl[0]) | ramrd | (ramwr & word) | w2;
	assign       ctrl   = {4'b0, ud_t_nxt, ld_t_nxt, ud_ce_nxt, ld_ce_nxt,
	                                   v_mem_ce, selio, addr[4:0], ctrl0};
	                                  // ctrl[0] pullup?

	always @(posedge clk or posedge rst) begin
		if (rst) begin
			addr <= 0;
			word <= 1;
			read <= 1;
			io <= 0;
			ramrd <= 1;
			ramwr <= 0;
			w1 <= 0;
			w2 <= 0;
			uxd_t <= 1;
			lxd_t <= 1;
			read_ub <=0;

		end
		else begin
			if (v_mem_ce) begin
				addr <= v_addr_nxt[4:0];
				word <= v_word_nxt;
				read <= v_read_nxt;
				io <= selio;
				read_ub <=  v_read_nxt & ~selio & ~v_addr_nxt[0] &
~v_word_nxt;
				ramrd <= v_read_nxt & ~selio;
				ramwr <= ~v_read_nxt & ~selio;
				uxd_t <= ~(v_read_nxt & v_dbus_nxt & ~selio &
v_word_nxt);
				lxd_t <= ~(v_read_nxt & v_dbus_nxt & ~selio);
			end
			w1 <= ~v_read_nxt & ~selio & v_mem_ce & ~v_word_nxt;
			w2 <= w1;
		end
	end

	// assign store_ub = ramwr_nxt & ~addr_nxt[0] & ~word_nxt;  // store
upper byte mux control
	assign store_ub = ramwr & ~addr[0] & ~word;  // store upper byte mux
control
	assign store_lb = ramwr & addr[0] & ~word;  // store lower byte mux
control

	//assign write_en_u = ramwr_nxt & (word_nxt | (~word_nxt &
~addr_nxt[0]));  // write upper byte if word or storing to high byte
	//assign write_en_l = ramwr_nxt & (word_nxt | (~word_nxt & addr_nxt[0]));
// write lower byte if word or storing to low byte

	assign write_en = (ramwr_nxt & v_word_nxt & v_mem_ce) | w1;  // write
upper byte if word or storing to high byte
```

```
	assign udlt_t = 1;  // not needed with 16-bit memory
	assign ld_t        = read;
	assign ud_t        = read;
	assign dma_req     = v_dma_req_in;
	assign zerodma     = v_zero_req_in;
	assign dma_ack     = v_dma & rdy;

endmodule

module ctrl_dec_syn(clk, rst, ctrl, sel, ud_t, ld_t, ud_ce, ld_ce, addr,
errorbus, id);
	input               clk;         // global clock
	input               rst;         // global async reset
	input  [47:0] ctrl;        // abstract control bus
	input  [2:0]        sel;         // peripheral select
	output              ud_t;        // active low read data MSB output
enable
	output              ld_t;        // active low read data LSB output
enable
	output              ud_ce;       // write data MSB clock enable
	output              ld_ce;       // write data LSB clock enable
	output [4:0]  addr;        // current I/O control register address
	input  [9:0] id;           // syndrome ID
	inout  [15:0] errorbus;  // global syndrome bus

	reg                        ud_t, ld_t, ud_ce, ld_ce;
	wor    [15:0] errorbus;

	// locals
	wire         ud_t_nxt, ld_t_nxt, ud_ce_nxt, ld_ce_nxt, mem_ce, selio;
	wire [4:0]   addr;
	wire   [15:0] v_ctrl;
	wire   v_sel;



	voter1nsyn  #(16) ctrl_v (.din(ctrl), .dout(v_ctrl), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID(id));
	voter1nsyn  #(1)  sel_v (.din(sel), .dout(v_sel), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID(id));

	assign { ud_t_nxt, ld_t_nxt, ud_ce_nxt, ld_ce_nxt, mem_ce, selio, addr }
		= v_ctrl[11:1];

	always @(posedge clk or posedge rst) begin
		if (rst) begin
			ud_ce <= 0;
			ld_ce <= 0;
			ud_t <= 1;
			ld_t <= 1;

		end
		else if (mem_ce) begin
			ud_ce <= v_sel & selio & ud_ce_nxt;
			ld_ce <= v_sel & selio & ld_ce_nxt;
			ud_t <= ~(v_sel & selio & ~ud_t_nxt);
			ld_t <= ~(v_sel & selio & ~ld_t_nxt);

		end
	end

endmodule
```

```
// original version as per Jan Gray below
/* memctrl.v -- XSOC memory/on-chip bus controller synthesizable Verilog model
 *
 * Copyright (C) 1999, 2000, Gray Research LLC.  All rights reserved.
 * The contents of this file are subject to the XSOC License Agreement;
 * you may not use this file except in compliance with this Agreement.
 * See the LICENSE file.
 *
 * $Header: /dist/xsocv/memctrl.v 6     4/06/00 10:55a Jan $
 * $Log: /dist/xsocv/memctrl.v $
 *
 * 6     4/06/00 10:55a Jan
 * polish

module memctrl(
	clk, rst,
	mem_ce, word_nxt, read_nxt, dbus_nxt, dma, addr_nxt,
	dma_req_in, zero_req_in,
	rdy, ud_t, ld_t, udlt_t, int_req, dma_req, zerodma,
	ram_oe_n, ram_we_n, xa_0, xdout_t, uxd_t, lxd_t,
	sel, ctrl, dma_ack);

	parameter           W      = 16;  // word width
	parameter           N      = W-1; // msb index
	parameter           XAN    = 16;  // external address msb index

	input               clk;          // global clock
	input               rst;          // global async reset
	input               mem_ce;             // memory access clock enable
	input               word_nxt;     // next access is word wide
	input               read_nxt;     // next access is read
	input               dbus_nxt;     // next access uses on-chip data bus
	input               dma;          // current access is a DMA transfer
	input  [N:0]  addr_nxt;     // address of next memory access
	input               dma_req_in;
	input               zero_req_in;
	output              rdy;          // current memory access is ready
	output              ud_t;         // active low store data MSB output
enable
	output              ld_t;         // active low store data LSB output
enable
	output              udlt_t;             // active low store data MSB->LSB
output enable
	output              int_req;      // interrupt request
	output              dma_req;      // DMA request
	output              zerodma;      // zero DMA counter request
	output              ram_oe_n;     // active low external RAM output enable
	output              ram_we_n;     // active low external RAM write enable
	output              xa_0;         // external RAM address lsb
	output              xdout_t;      // active low external RAM data output
enable
	output              uxd_t;        // active low external RAM MSB input
output en
	output              lxd_t;        // active low external RAM LSB input
output en
	output [7:0]  sel;          // on-chip peripheral select
	output [15:0] ctrl;         // abstract control bus
	output              dma_ack;      // DMA ackwnowledge

	reg                        uxd_t, lxd_t;
```

```
// locals
wire         selio  = addr_nxt[15:8] == 8'hFF;
assign       sel[0] = addr_nxt[7:5] == 0;
assign       sel[1] = addr_nxt[7:5] == 1;
assign       sel[2] = addr_nxt[7:5] == 2;
assign       sel[3] = addr_nxt[7:5] == 3;
assign       sel[4] = addr_nxt[7:5] == 4;
assign       sel[5] = addr_nxt[7:5] == 5;
assign       sel[6] = addr_nxt[7:5] == 6;
assign       sel[7] = addr_nxt[7:5] == 7;
wire         ud_t_nxt     = ~(word_nxt & read_nxt);
wire         ld_t_nxt     = ~read_nxt;
wire         ud_ce_nxt    = word_nxt & ~read_nxt;
wire         ld_ce_nxt    = ~read_nxt;
reg [4:0]    addr;  // current access address
reg                 word;  // current access is word wide
reg                 read;  // current access is read
reg                 io;          // current access is to I/O space

// memory / I/O access FSM
reg                 ramrd; // RAM read access
reg                 ramwr; // RAM write access
reg                 w12;   // RAM write half-clock states W1, W2
reg                 w34;   // RAM write half-clock states W3, W4
reg                 w56;   // RAM write half-clock states W5, W6
reg                 w23_45;      // RAM write half-clock states W2, W3,
and W4, W5
reg                 w45;   // RAM write half-clock states W4, W5

assign       rdy          = (io&ctrl[0]) | ramrd | (w34&~word) | w56;
assign       ctrl   = {4'b0, ud_t_nxt, ld_t_nxt, ud_ce_nxt, ld_ce_nxt,
                               mem_ce, selio, addr[4:0], 1'b1};
                              // ctrl[0] pullup?

always @(posedge clk or posedge rst) begin
       if (rst) begin
              addr <= 0;
              word <= 1;
              read <= 1;
              io <= 0;
              ramrd <= 1;
              ramwr <= 0;
              w12 <= 0;
              w34 <= 0;
              w56 <= 0;
              uxd_t <= 1;
              lxd_t <= 1;
       end
       else begin
              if (mem_ce) begin
                     addr <= addr_nxt[4:0];
                     word <= word_nxt;
                     read <= read_nxt;
                     io <= selio;
                     ramrd <= read_nxt & ~selio;
                     ramwr <= ~read_nxt & ~selio;
                     uxd_t <= ~(read_nxt & dbus_nxt & ~selio & word_nxt);
                     lxd_t <= ~(read_nxt & dbus_nxt & ~selio);
              end
              w12 <= ~read_nxt & ~selio & mem_ce;
              w34 <= w12;
```

```
                w56 <= w34 & word;
            end
        end
        always @(negedge clk or posedge rst) begin
            if (rst) begin
                w23_45 <= 0;
                w45 <= 0;
            end
            else begin
                w23_45 <= w12 | (w34 & word);
                w45 <= w34 & word;
            end
        end

        assign xdout_t      = ~(w23_45 | w56);
        assign ram_we_n     = ~(w23_45 & ~w34);
        assign ram_oe_n     = ramwr;
        assign xa_0         = word ? ~(ramwr ? (w45|w56) : clk) : addr[0];
        assign udlt_t = ~((w45|w56) & ~read);
        assign ld_t         = ~(~(w45|w56) & ~read);
        assign ud_t         = read;
        assign dma_req      = dma_req_in;
        assign zerodma      = zero_req_in;
        assign dma_ack      = dma & rdy;
        assign int_req      = 0;

endmodule


// ctrl_dec -- XSOC abstract control bus decoder

module ctrl_dec(clk, rst, ctrl, sel, ud_t, ld_t, ud_ce, ld_ce, addr);
        input               clk;        // global clock
        input               rst;        // global async reset
        input  [15:0] ctrl;         // abstract control bus
        input               sel;        // peripheral select
        output              ud_t;       // active low read data MSB output
enable
        output              ld_t;       // active low read data LSB output
enable
        output              ud_ce;      // write data MSB clock enable
        output              ld_ce;      // write data LSB clock enable
        output [4:0]  addr;         // current I/O control register address

        reg                         ud_t, ld_t, ud_ce, ld_ce;

        // locals
        wire          ud_t_nxt, ld_t_nxt, ud_ce_nxt, ld_ce_nxt, mem_ce, selio;
        wire [4:0]    addr;

        assign { ud_t_nxt, ld_t_nxt, ud_ce_nxt, ld_ce_nxt, mem_ce, selio, addr }
            = ctrl[11:1];

        always @(posedge clk or posedge rst) begin
            if (rst) begin
                ud_ce <= 0;
                ld_ce <= 0;
                ud_t <= 1;
                ld_t <= 1;
            end
            else if (mem_ce) begin
                ud_ce <= sel & selio & ud_ce_nxt;
```

```
                    ld_ce <= sel & selio & ld_ce_nxt;
                    ud_t <= ~(sel & selio & ~ud_t_nxt);
                    ld_t <= ~(sel & selio & ~ld_t_nxt);
              end
       end

endmodule

*/
```

J. MEMIO.V

```
`timescale 1ns / 1ps
module memio(sub, slb, rub, din, eccdout, mem_out, xd, eccdin, syndrome);
       input sub;          // store upper byte
       input slb;          // store lower byte
       input rub;          // read upper byte
       input [15:0] din;          // MDR input
       input  [21:0] eccdout;

       output [15:0]       mem_out;            // CPU data bus

       output [15:0]       xd;    // external data bus to instruction register
       output [21:0]       eccdin;
       output [5:0]  syndrome;

       wire [7:0]          datainusb,datainlsb;
       wire [7:0]          dataoutusb, dataoutlsb;
       wire [15:0]         decodeout;

       eccencode eccencode1 (.din({datainusb,datainlsb}), .dout(eccdin));
       eccdecode eccdecode1 (.din(eccdout),
.dout(decodeout),.syndrome(syndrome));

       assign xd[15:0] = decodeout[15:0];

       // data in is input into the registers at the BRAM interface and is not
clocked

       assign datainlsb = sub ? decodeout[7:0] : din[7:0];
       assign datainusb = slb ? decodeout[15:8] : (sub ? din[7:0] : din[15:8]);

       // data out of the memory is on the next clock cycle;  output control
signals should be registered
       // for proper operation

       assign dataoutlsb = rub ? decodeout[15:8] : decodeout[7:0];
       assign dataoutusb = decodeout[15:8];

       assign mem_out[15:0] = {dataoutusb[7:0],dataoutlsb[7:0]};

endmodule
```

K. PC104QUEUE.V

```
`timescale 1ns / 1ps
```

```
//     Created 08/01/2008 by David Dwiggins Jr
// synthetic RAM created for pc/104 output queue
// created to avoid using distributed RAM elements
// synchronous write, async read
// byte wide 16 entries
module qram_8x16(clk, rst, readaddr, writeaddr, we, din, dout);
       input              clk;
       input              rst;
       input  [3:0]  readaddr;           // Read address
       input [3:0]  writeaddr;     // Write address
       input                     we;                         // Write enable
       input [7:0]  din;                        // Data in
       output [7:0] dout;                       // Data out

       wire   [15:0]       readdrive;          // output enables on each entry
       wire   [15:0]       writedrive;
       wire   [15:0] writeendrive;// write enables for entries

       decode4 read_decode(.din(readaddr), .dout(readdrive));
       decode4 write_decode(.din(writeaddr), .dout(writedrive));

       assign writeendrive = we ? writedrive : 16'b0;

       genvar i;

       generate for (i=0; i<16; i=i+1) begin: myram_loop
              // instantiate each queue slot
              qd8 loop_d8(.clk(clk), .ce(writeendrive[i]), .clr(rst), .din(din),
.ddrive(readdrive[i]),.dout(dout));
       end
       endgenerate

endmodule

// qd8 - 8 bit D positive edge-clocked flip flop
module qd8(clk, ce, clr, din, ddrive, dout);
       input                     clk;
       input              ce;
       input              clr;
       input [7:0]  din;
       input              ddrive;
       output [7:0] dout;

       reg    [7:0]        data;
       tri    [7:0]        dout;

       always @(posedge clk or posedge clr) begin
              if (clr)
                     data <= 0;
              else if (ce)
                     data <= din;
       end

       assign dout = (ddrive) ? data : 8'bz;

endmodule

// PC104 Interface Output Queue
// 16-entry queue
// Separate clock domains on read and write
// Shared variable remove request, set on remove clock positive edge and cleared
// on read process by system clock
```

```
// Shared variable implemented with a flip-flop with an async clear
// queue is not write-through, must have a read for first data to appear on
output

module pc104queue_8x16(din, clk, rst, add, remove, dout,
                                              empty, nearlyfull, full,
remove_clk);
   input [23:0]           din;                        // Queue data in
   input                       clk;                        // System clock
   input                       rst;                        // Global Reset
      input [2:0]              add;                        // Add request
      input [2:0]              remove;                     // Remove request,
remove_clk clocked
   output [7:0]           dout;                       // Queue data out
   output                      empty;                 // 1 if empty
      output                        nearlyfull;           // 1 if 15 entries
in queue
   output                      full;                       // 1 if full
      input                    remove_clk;           //     Clock for remove
signal

      wire [7:0]               queue_data;
      reg [4:0]                count;
      reg [3:0]                first;
      reg [3:0]                store_nxt;
      reg                           remove_state0;  // no remove pending
      reg                           remove_state1;  // just got a remove,
process

      reg [7:0]                dout;
      wire                          clocked_remove;

      wire                          we;                               //
queue memory write enable

      wire                          countisone;
      wire                          nearlyfull;
      wire   [7:0]             v_din;
      wire                          v_add;
      wire                          v_remove;

      voter1n #(8) din_v (.din(din), .dout(v_din));
      voter1n #(1) add_v (.din(add), .dout(v_add));
      voter1n #(1) remove_v (.din(remove), .dout(v_remove));

      assign empty = ~(|count);                     // empty if no count bits
set
      assign nearlyfull = &(count[3:0]);      // 15 = last 4 bits all on
      assign full = count[4];                            // 16 = bit 5 on

      qram_8x16 ram1(.clk(clk), .rst(rst), .readaddr(first),
.writeaddr(store_nxt), .we(we), .din(v_din), .dout(queue_data));

      assign we = v_add & ~(full);

      FDC read_request (.Q(clocked_remove),
                                              .C(remove_clk),
                                              .CLR(remove_state1),
                                              .D(v_remove));

      always @(posedge clk or posedge rst) begin
            if(rst) begin
```

```
			count<=0;
			first<=0;
			store_nxt<=0;
			remove_state0 <= 1;
			remove_state1 <= 0;
			dout <= 0;
		end
		else begin
			// got a remove, process
			if (clocked_remove & remove_state0) begin
				remove_state0 <= 0;
				remove_state1 <= 1;
			end
			// clearing the read FF
			if (remove_state1) begin
				remove_state1 <= 0;
				remove_state0 <= 1;
			end

			// add
			if ((v_add & ~clocked_remove) | (v_add & clocked_remove &
~remove_state0)) begin
				if (~full) begin
					count<=count+1;
					store_nxt<=store_nxt+1;
				end
			end
			// remove
			else if (~v_add & clocked_remove & remove_state0) begin
				if(empty) begin
					dout <= 0;
				end
				else begin
					count<=count-1;
					first<=first+1;
					dout <= queue_data;
				end
			end
			// add & remove
			else if (v_add & clocked_remove & remove_state0) begin
				if(empty) begin
					count<=count+1;
					store_nxt<=store_nxt+1;
					dout <= 0;
				end
				else if (full) begin
					count<=count-1;
					first<=first+1;
					dout <= queue_data;
				end
				else begin
					store_nxt<=store_nxt+1;
					first<=first+1;
					dout <= queue_data;
				end
			end
		end
	end

endmodule
```

L. PCFILE.V

```
`timescale 1ns / 1ps
module d16x1out(clk, ce, clr, din, dout);
	input clk;
	input ce;
	input clr;
	input [15:0] din;
	output [15:0] dout;

	reg [15:0] dout;

	always @(posedge clk) begin
		if (clr)
			dout <= 0;
		else
			if (ce)
			dout <= din;
	end


endmodule


module pcfile(clk,rst,we,addr,din,adout);
	input clk;
	input rst;
	input we;
	input  addr;
	input [15:0] din;
	output [15:0] adout;

	wire [15:0] reg0out;
		wire [15:0] reg1out;
		wire reg0load;
		wire reg1load;

		assign reg0load = we & ~addr;
		assign reg1load = we & addr;

		d16x1out reg0(.clk(clk), .ce(reg0load), .clr(rst), .din(din),
.dout(reg0out));
		d16x1out reg1(.clk(clk), .ce(reg1load), .clr(rst), .din(din),
.dout(reg1out));

		assign adout = addr ? reg1out : reg0out;

endmodule
```

M. QUEUE1.V

```
`include "voterids.v"
`timescale 1ns / 1ps
```

```
module ram_16x8(clk, rst, readaddr, writeaddr, we, din, dout);
       input clk;
       input rst;
       input [2:0] readaddr;
       input [2:0] writeaddr;
       input we;
       input [15:0] din;
       output [15:0] dout;

       wire   [15:0] readdrive;
       wire   [15:0] writedrive;
       wire   [15:0] writeendrive;

       decode4 read_decode(.din({1'b0,readaddr}), .dout(readdrive[7:0]));
       decode4 write_decode(.din({1'b0,writeaddr}), .dout(writedrive[7:0]));

       assign writeendrive = we ? writedrive : 8'b0;

       genvar i;

       generate for (i=0; i<8; i=i+1) begin: myram_loop
              d16 loop_d16(.clk(clk), .ce(writeendrive[i]), .clr(rst),
.din(din), .ddrive(readdrive[i]),.dout(dout));
       end
       endgenerate

endmodule

module d16(clk, ce, clr, din, ddrive, dout);
       input clk;
       input ce;
       input clr;
       input [15:0] din;
       input ddrive;
       output [15:0] dout;

       reg [15:0] data;
       tri [15:0] dout;

       always @(posedge clk or posedge clr) begin
              if (clr)
                     data <= 0;
              else if (ce)
                     data <= din;
       end

       assign dout = (ddrive) ? data : 16'bz;
endmodule

module queue_16x8(din, clk, rst, add, remove, dout, empty, nearlyfull, full,
errorbus);
    input [15:0] din;
    input                  clk;
    input                  rst;
        input [2:0]        add;
        input [2:0] remove;
    output [15:0] dout;
    output                 empty;
        output                   nearlyfull;
    output                 full;
        inout [15:0] errorbus;
```

```
        wire [15:0] queue_data;

        reg [3:0]          count;
        reg [2:0]          first;
    reg [2:0]              store_nxt;
        wire [15:0] dout;
        reg                       outputdata_nxt;
        wor [15:0]         errorbus;
        wire               we;
        wire               countisone;
        wire               nearlyfull;
        wire                      v_add;
        wire               v_remove;

        assign empty = ~(|count);
        assign countisone = (count==1);
        assign nearlyfull = &(count[2:0]);
        assign full = count[3];

       ram_16x8 ram1(.clk(clk), .rst(rst), .readaddr(first),
.writeaddr(store_nxt), .we(we), .din(din), .dout(queue_data));

       voter1nsyn #(1)     add_v (.din(add), .dout(v_add), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID(`ER_QU_ADD_V));
       voter1nsyn #(1)     remove_v (.din(remove), .dout(v_remove),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID(`ER_QU_REM_V));

       assign we = v_add & ~(full);

       assign dout = outputdata_nxt ? queue_data : 8'b0;

       always @(posedge clk or posedge rst) begin
              if(rst) begin
                     count<=0;
                     first<=0;
                     store_nxt<=0;
                     outputdata_nxt<=0;
              end
              else begin
                     // add
                     if (v_add & ~v_remove) begin
                            if(empty) begin
                                   count<=count+1;
                                   store_nxt<=store_nxt+1;
                                   outputdata_nxt<=1;
                            end
                            else if (~full) begin
                                   count<=count+1;
                                   store_nxt<=store_nxt+1;
                            end
                     end
                     // remove
                     else if (~v_add & v_remove) begin
                            if (~empty) begin
                                   if(countisone) begin
                                          outputdata_nxt<=0;
                                          count<=count-1;
                                          first<=first+1;
                                   end
                                   else begin
                                          count<=count-1;
                                          first<=first+1;
```

```
                        end
                    end
                end
                // add & remove
                else if (v_add & v_remove) begin
                    if(empty) begin  // only add
                        count<=count+1;
                        store_nxt<=store_nxt+1;
                        outputdata_nxt<=1;
                    end
                    else if (full) begin  // only remove
                        count<=count-1;
                        first<=first+1;
                    end
                    else begin
                        store_nxt<=store_nxt+1;
                        first<=first+1;
                    end
                end
            end
        end

endmodule
```

N. REGISTER16X16.V

```
`timescale 1ns / 1ps
module zerox2out(adrive, bdrive, aout, bout);
    input adrive;
    input bdrive;
    output [15:0] aout,bout;

    tri [15:0] aout,bout;

    assign aout = (adrive) ? 16'b0 : 16'bz;
    assign bout = (bdrive) ? 16'b0 : 16'bz;

endmodule

module d16x2out(clk, ce, clr, din, adrive, bdrive, aout, bout);
    input clk;
    input ce;
    input clr;
    input [15:0] din;
    input adrive;
    input bdrive;
    output [15:0] aout,bout;

    reg [15:0] dout;
    tri [15:0] aout,bout;


    always @(posedge clk or posedge clr) begin
        if (clr)
            dout <= 0;
        else if (ce)
            dout <= din;
    end

    assign aout = (adrive) ? dout : 16'bz;
```

```
       assign bout = (bdrive) ? dout : 16'bz;

endmodule

module decode4(din, dout);
       input [3:0] din;
       output [15:0] dout;

       reg [15:0] dout;

       always @(din) begin
              case (din)
                     4'h0 : dout <= 16'h0001;
                     4'h1 : dout <= 16'h0002;
                     4'h2 : dout <= 16'h0004;
                     4'h3 : dout <= 16'h0008;
                     4'h4 : dout <= 16'h0010;
                     4'h5 : dout <= 16'h0020;
                     4'h6 : dout <= 16'h0040;
                     4'h7 : dout <= 16'h0080;
                     4'h8 : dout <= 16'h0100;
                     4'h9 : dout <= 16'h0200;
                     4'hA : dout <= 16'h0400;
                     4'hB : dout <= 16'h0800;
                     4'hC : dout <= 16'h1000;
                     4'hD : dout <= 16'h2000;
                     4'hE : dout <= 16'h4000;
                     4'hF : dout <= 16'h8000;
              endcase
       end
endmodule


module register16x16(clk,rst,we,addra,addrb,addrw,din,adout,bdout);
    input clk;
    input rst;
    input we;
    input [3:0] addra,addrb,addrw;
    input [15:0] din;
    output [15:0] adout,bdout;


    wire [15:0] adrive,bdrive;
    wire [15:0] wdecode;
    wire [15:0] regwrite;

        genvar i;


       decode4 rega_decode(addra,adrive);
       decode4 regb_decode(addrb,bdrive);
       decode4 regw_decode(addrw,wdecode);

       assign regwrite = we ? wdecode : 16'b0;

       zerox2out zero0(adrive[0], bdrive[0], adout, bdout);

       generate for (i=1;i<16;i=i+1) begin: regg_loop
              d16x2out reg0(clk, regwrite[i], rst, din, adrive[i], bdrive[i],
adout, bdout);
       end
       endgenerate
```

```
endmodule
```

O. SELECTMAP_CONFIG_XSOC.V

```
`timescale 1ns / 1ps
`include "voterids.v"
`define      SELECTMAP_CF_REQUEST                    5'h00
`define      SELECTMAP_CF_FLASH_BASE_LOW      5'h02
`define      SELECTMAP_CF_FLASH_BASE_HIGH     5'h04
`define      SELECTMAP_CF_STATUS                     5'h06
// change the next line to voter1nsyn to reenable error bus reporting
// on this module
`define      SMC_VOTER                                               voter1nsynh
module selectmap_config_xsoc_mod(clk, rst, ld_t, ld_ce, addr,
                          d, dout,     errorbus,
                          T_SELECTMAP_INIT_o, T_SELECTMAP_WRITE_o,
                          T_SELECTMAP_CS_o, T_SELECTMAP_DATA_o,
                          T_FLASH_DATA_i, T_FLASH_ADDRESS_o,
                          SM_CONFIG_STATUS_o);

      input                      clk;
      input                      rst;
      input  [2:0]         ld_t;
      input  [2:0]         ld_ce;
      input [14:0] addr;
      input  [47:0] d;
      output               dout;
      inout  [15:0] errorbus;
      output               T_SELECTMAP_INIT_o;
      output               T_SELECTMAP_WRITE_o;
      output               T_SELECTMAP_CS_o;
      output [7:0]  T_SELECTMAP_DATA_o;
      input [15:0]  T_FLASH_DATA_i;
      output[21:0]  T_FLASH_ADDRESS_o;
      output               SM_CONFIG_STATUS_o;



      wor [15:0]           errorbus;
      reg                        dout;
      reg    [4:0]         flashhi;     // high word flash base
      reg    [15:0] flashlo;     // low word flash base
      reg                        conf_req;// configure request

      wire   [15:0] v_d;         // voted data bus
      wire   [4:0]         v_addr;      // voted address
      wire                       v_ld_ce;
      wire                       v_ld_t;
      wire                       SM_CONFIG_STATUS_o;


      `SMC_VOTER #(16)    d_v (.din(d), .dout(v_d), .errorbus(errorbus),
.clk(clk), .rst(rst), .ID(`SMC_D));
      `SMC_VOTER #(5)     addr_v (.din(addr), .dout(v_addr),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID(`SMC_ADDR));
```

```
	`SMC_VOTER #(1)	ld_t_v (.din(ld_t), .dout(v_ld_t),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID(`SMC_LD_T));

	`SMC_VOTER #(1)	ld_ce_v (.din(ld_ce), .dout(v_ld_ce),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID(`SMC_LD_CE));


	always @(posedge clk or posedge rst) begin
		if (rst)
			begin
				flashhi <= 0;
				flashlo <= 0;
				conf_req <= 0;
			end
		else begin
			// I got your request, working on it, clear request
			if (SM_CONFIG_STATUS_o & conf_req)
				begin
					conf_req <= 1'b0;
				end
			// writing, addr 0 = low flash base, addr 2 = high flash
base, addr 4 = conf request
			if (v_ld_ce)
				begin
					if (v_addr==`SELECTMAP_CF_FLASH_BASE_LOW)
						begin
							flashlo <= v_d[15:0];
						end
					else if
(v_addr==`SELECTMAP_CF_FLASH_BASE_HIGH)
						begin
							flashhi <= v_d[4:0];
						end
					else if (v_addr==`SELECTMAP_CF_REQUEST)
						begin
							conf_req <= 1'b1;
						end
				end
			else if (v_ld_t)
				begin
					if(v_addr==`SELECTMAP_CF_STATUS) begin
						dout <= SM_CONFIG_STATUS_o;
					end
				end
		end
	end


	selectmap_config selectmap_config0(	.T_CLOCK_i(clk),

	.RESET_i(rst),

	.T_SELECTMAP_INIT_o(T_SELECTMAP_INIT_o),

	.T_SELECTMAP_WRITE_o(T_SELECTMAP_WRITE_o),

	.T_SELECTMAP_CS_o(T_SELECTMAP_CS_o),

	.T_SELECTMAP_DATA_o(T_SELECTMAP_DATA_o),

	.T_FLASH_DATA_i(T_FLASH_DATA_i),
```

```
       .T_FLASH_ADDRESS_o(T_FLASH_ADDRESS_o),

       .SM_CONFIG_RQST_i(conf_req),

       .SM_CONFIG_STATUS_o(SM_CONFIG_STATUS_o),

       .SM_FLASH_BASE_i({flashhi[4:0],flashlo[15:0]}));



endmodule

module selectmap_config_xsoc(clk, rst, ctrl, sel, d, errorbus,
                           T_SELECTMAP_INIT_o, T_SELECTMAP_WRITE_o,
                           T_SELECTMAP_CS_o, T_SELECTMAP_DATA_o,
                           T_FLASH_DATA_i, T_FLASH_ADDRESS_o,
                           SM_CONFIG_STATUS_o);

   input                   clk;
   input                   rst;
   input [47:0]     ctrl;
   input [2:0]      sel;
   inout [47:0]     d;
       inout [15:0] errorbus;
       output              T_SELECTMAP_INIT_o;
       output              T_SELECTMAP_WRITE_o;
       output              T_SELECTMAP_CS_o;
       output [7:0] T_SELECTMAP_DATA_o;
       input [15:0] T_FLASH_DATA_i;
       output[21:0] T_FLASH_ADDRESS_o;
       output              SM_CONFIG_STATUS_o;

       wire [2:0]          SELECTMAP_INIT_o;
       wire [2:0]          SELECTMAP_WRITE_o;
       wire [2:0]          SELECTMAP_CS_o;
       wire [23:0]         SELECTMAP_DATA_o;
       wire [65:0]         FLASH_ADDRESS_o;
       wire [2:0]          CONFIG_STATUS;

       wire                       clk;         // Clock
       wire                       rst;         // Reset
       wire   [47:0] ctrl;        //     16-bit memory control bus
       wire   [2:0]        sel;         // 1-bit peripheral select # (IO Base
Select)
                                                             // I/O 0 is at base
address FF00, subsequent IDs
                                                             //     are each 20h
higher
       tri    [47:0] d;                 // 16-bit tri-state data bus
       wor    [15:0]       errorbus;//  16-bit Global Error Bus, wired-OR

       wire   [14:0] addr;        // 5-bit Peripheral Address Space
       wire   [2:0]        ud_t;        // Active-low upper byte data bus tri-
state drive enable
       wire   [2:0]        ld_t;        // Active-low lower byte data bus tri-
state drive enable
       wire   [2:0]        ud_ce; // Active-high upper byte valid data clock
enable
       wire   [2:0]        ld_ce; // Active-high lower byte valid data clock
enable
```

```
	wire 			SM_CONFIG_STATUS_o;

	wire	[2:0]	dout;

	ctrl_dec_syn dec0(.clk(clk), .rst(rst), .ctrl(ctrl), .sel(sel),
.ud_t(ud_t[0]),
			.ld_t(ld_t[0]), .ud_ce(ud_ce[0]), .ld_ce(ld_ce[0]),
.addr(addr[4:0]),
			.errorbus(errorbus), .id(`SMC_CTRL_DEC));

	ctrl_dec_syn dec1(.clk(clk), .rst(rst), .ctrl(ctrl), .sel(sel),
.ud_t(ud_t[1]),
			.ld_t(ld_t[1]), .ud_ce(ud_ce[1]), .ld_ce(ld_ce[1]),
.addr(addr[9:5]),
			.errorbus(errorbus), .id(`SMC_CTRL_DEC));


	ctrl_dec_syn dec2(.clk(clk), .rst(rst), .ctrl(ctrl), .sel(sel),
.ud_t(ud_t[2]),
			.ld_t(ld_t[2]), .ud_ce(ud_ce[2]), .ld_ce(ld_ce[2]),
.addr(addr[14:10]),
			.errorbus(errorbus), .id(`SMC_CTRL_DEC));

	`SMC_VOTER #(1)	SELECTMAP_INIT_v (.din(SELECTMAP_INIT_o),
.dout(T_SELECTMAP_INIT_o),
				.errorbus(errorbus), .clk(clk),
.rst(rst), .ID(`SMC_SELECTMAP_INIT));
	`SMC_VOTER #(1)	SELECTMAP_WRITE_v (.din(SELECTMAP_WRITE_o),
.dout(T_SELECTMAP_WRITE_o),
				.errorbus(errorbus), .clk(clk),
.rst(rst), .ID(`SMC_SELECTMAP_WRITE));
	`SMC_VOTER #(1)	SELECTMAP_CS_v (.din(SELECTMAP_CS_o),
.dout(T_SELECTMAP_CS_o),
				.errorbus(errorbus), .clk(clk),
.rst(rst), .ID(`SMC_SELECTMAP_CS));
	`SMC_VOTER #(8)	SELECTMAP_DATA_v (.din(SELECTMAP_DATA_o),
.dout(T_SELECTMAP_DATA_o),
				.errorbus(errorbus), .clk(clk),
.rst(rst), .ID(`SMC_SELECTMAP_DATA));
	`SMC_VOTER #(22)	FLASH_ADDRESS_v (.din(FLASH_ADDRESS_o),
.dout(T_FLASH_ADDRESS_o),
				.errorbus(errorbus), .clk(clk),
.rst(rst), .ID(`SMC_FLASH_ADDR));
	`SMC_VOTER #(1)	CONFIG_STATUS_v (.din(CONFIG_STATUS),
.dout(SM_CONFIG_STATUS_o),
				.errorbus(errorbus), .clk(clk),
.rst(rst), .ID(`SMC_CONFIG_STATUS));

	selectmap_config_xsoc_mod sm_mod0(	.clk(clk),

	.rst(rst),

	.ld_t(ld_t), .ld_ce(ld_ce), .addr(addr),

	.d(d), .dout(dout[0]),

	.T_SELECTMAP_INIT_o(SELECTMAP_INIT_o[0]),

	.T_SELECTMAP_WRITE_o(SELECTMAP_WRITE_o[0]),

	.T_SELECTMAP_CS_o(SELECTMAP_CS_o[0]),
```

```
.T_SELECTMAP_DATA_o(SELECTMAP_DATA_o[7:0]),
.T_FLASH_DATA_i(T_FLASH_DATA_i),
.T_FLASH_ADDRESS_o(FLASH_ADDRESS_o[21:0]),
.SM_CONFIG_STATUS_o(CONFIG_STATUS[0]),
.errorbus(errorbus));
selectmap_config_xsoc_mod sm_mod1(      .clk(clk),
.rst(rst),
.ld_t(ld_t), .ld_ce(ld_ce), .addr(addr),
.d(d), .dout(dout[1]),
.T_SELECTMAP_INIT_o(SELECTMAP_INIT_o[1]),
.T_SELECTMAP_WRITE_o(SELECTMAP_WRITE_o[1]),
.T_SELECTMAP_CS_o(SELECTMAP_CS_o[1]),
.T_SELECTMAP_DATA_o(SELECTMAP_DATA_o[15:8]),
.T_FLASH_DATA_i(T_FLASH_DATA_i),
.T_FLASH_ADDRESS_o(FLASH_ADDRESS_o[43:22]),
.SM_CONFIG_STATUS_o(CONFIG_STATUS[1]),
.errorbus(errorbus));
selectmap_config_xsoc_mod sm_mod2(      .clk(clk),
.rst(rst),
.ld_t(ld_t), .ld_ce(ld_ce), .addr(addr),
.d(d), .dout(dout[2]),
.T_SELECTMAP_INIT_o(SELECTMAP_INIT_o[2]),
.T_SELECTMAP_WRITE_o(SELECTMAP_WRITE_o[2]),
.T_SELECTMAP_CS_o(SELECTMAP_CS_o[2]),
.T_SELECTMAP_DATA_o(SELECTMAP_DATA_o[23:16]),
.T_FLASH_DATA_i(T_FLASH_DATA_i),
.T_FLASH_ADDRESS_o(FLASH_ADDRESS_o[65:44]),
.SM_CONFIG_STATUS_o(CONFIG_STATUS[2]),
.errorbus(errorbus));

assign d[7:0]              = ld_t[0] ? 8'bz : {7'b0, dout[0]};
```

```
      assign d[15:8]             = ud_t[0] ? 8'bz : 8'b0;
      assign d[23:16]      = ld_t[1] ? 8'bz : {7'b0, dout[1]};
      assign d[31:24]      = ud_t[1] ? 8'bz : 8'b0;
      assign d[39:32]      = ld_t[2] ? 8'bz : {7'b0, dout[2]};
      assign d[47:40]      = ud_t[2] ? 8'bz : 8'b0;

endmodule
```

P. SELECTMAP_RB_XSOC.V

```
`include "voterids.v"
`timescale 1ns / 1ps
// change so reading Error Word Addr clears status
// change so scrub halts when error occurs
`define      SM_RB_RQST_ADDR                              5'h0
`define      SM_RB_STATUS_ADDR                            5'h1
`define      SM_RB_READ_DATA_ADDR                  5'h2
`define  SM_RB_ERROR_LOC_H_ADDR               5'h4
`define  SM_RB_ERROR_LOC_L_ADDR               5'h6
`define      SM_RB_ERROR_WORD_ADDR                 5'h8
`define      SM_RB_DATA_RDY_ADDR                   5'hA
`define      SM_RB_FLASH_BASE_LOW                  5'hC
`define      SM_RB_FLASH_BASE_HIGH                 5'hE
// change the next line to read voter1nsyn to
// reenable error reporting for this module's voters
`define      SMR_VOTER                                           voter1nsynh

module selectmap_rb_xsoc_mod(clk, rst, ld_t, ld_ce, addr,
                           d, dout, errorbus, int_req,
                           T_SELECTMAP_INIT_o, T_SELECTMAP_WRITE_o,
                           T_SELECTMAP_CS_o, T_SELECTMAP_DATA_o,
                           T_SELECTMAP_DATA_i,
                           T_FLASH_DATA_i, T_FLASH_ADDRESS_o,
                           SM_RB_STATUS_o, T_CCLK_o);

      input              clk;
   input                 rst;
      input  [2:0]       ld_t;
      input  [2:0]       ld_ce;
      input [14:0] addr;
   input [47:0]     d;
      output [15:0]dout;
      inout [15:0] errorbus;
      output             int_req;
      output             T_SELECTMAP_INIT_o;
      output             T_SELECTMAP_WRITE_o;
      output             T_SELECTMAP_CS_o;
      output [7:0] T_SELECTMAP_DATA_o;
      input [7:0]        T_SELECTMAP_DATA_i;
      input [15:0] T_FLASH_DATA_i;
      output[21:0] T_FLASH_ADDRESS_o;
      output             SM_RB_STATUS_o;
      output             T_CCLK_o;

      wor    [15:0]      errorbus;//  16-bit Global Error Bus, wired-OR

      wire               v_ld_t;             // Active-low lower byte data bus
tri-state drive enable
```

```
	wire			v_ld_ce;	//  Active-high  lower  byte  valid  data
clock enable
	wire	[15:0] v_d;		// voted data bus
	wire	[4:0]		v_addr;		// voted address

   wire				SM_RB_STATUS_o;
	wire [7:0]		SM_RB_READ_DATA_o;
	wire [7:0]	SM_RB_ERROR_LOC_H_o;
	wire [15:0]		SM_RB_ERROR_LOC_L_o;
	wire [15:0]		SM_RB_ERROR_WORD_o;
	wire				SM_RB_DATA_RDY_o;
	reg				SM_RB_ACK_i;

	reg				rb_req;// configure request
	reg [15:0]		flash_addr_low;
	reg [4:0]		flash_addr_high;

	reg	[15:0] dout;		// 16-bit data bus

	assign errorbus = 16'b0;
	assign int_req = 1'b0;

	`SMR_VOTER #(16)	d_v (.din(d), .dout(v_d), .errorbus(errorbus),
			.clk(clk), .rst(rst), .ID(`SMR_D));

	`SMR_VOTER #(5)	addr_v		(.din(addr),		.dout(v_addr),
.errorbus(errorbus),
			.clk(clk), .rst(rst), .ID(`SMR_ADDR));

	`SMR_VOTER #(1)	ld_t_v		(.din(ld_t),		.dout(v_ld_t),
.errorbus(errorbus),
			.clk(clk), .rst(rst), .ID(`SMR_LD_T));

	`SMR_VOTER #(1)	ld_ce_v		(.din(ld_ce),		.dout(v_ld_ce),
.errorbus(errorbus),
			.clk(clk), .rst(rst), .ID(`SMR_LD_CE));


	always @(posedge clk or posedge rst) begin
		if (rst)
			begin
				rb_req <= 1'b0;
				flash_addr_high <= 5'b00000;
				flash_addr_low <= 16'b0000000001001000;
			end
		else begin
			// I got your request, working on it, clear request
			if (SM_RB_STATUS_o & rb_req)
				begin
					rb_req <= 1'b0;
				end
			//
			if (SM_RB_ACK_i & ~SM_RB_DATA_RDY_o)
				begin
					SM_RB_ACK_i <= 1'b0;
				end
			// writing, addr 0 = low flash base, addr 2 = high flash
base, addr 4 = conf request
			if (v_ld_ce)
				begin
					if (v_addr==`SM_RB_RQST_ADDR)
						begin
```

```
                                        rb_req <= 1'b1;
                                end
                        if (v_addr==`SM_RB_FLASH_BASE_HIGH)
                                begin
                                        flash_addr_high <= v_d[4:0];
                                end
                        if (v_addr==`SM_RB_FLASH_BASE_LOW)
                                begin
                                        flash_addr_low <= v_d;
                                end

                end
        else if (v_ld_t)
                begin
                        case (v_addr)
                        `SM_RB_STATUS_ADDR:
                                dout <= {15'b0, SM_RB_STATUS_o};
                        `SM_RB_READ_DATA_ADDR:
                                dout <= {8'b0, SM_RB_READ_DATA_o};
                        `SM_RB_ERROR_LOC_H_ADDR:
                                dout <= {8'b0, SM_RB_ERROR_LOC_H_o};
                        `SM_RB_ERROR_LOC_L_ADDR:
                                dout <= SM_RB_ERROR_LOC_L_o;
                        `SM_RB_ERROR_WORD_ADDR:
                                dout <= SM_RB_ERROR_WORD_o;
                                SM_RB_ACK_i <= 1'b1;  // error word read
is an ack of getting all the data
                        `SM_RB_DATA_RDY_ADDR:
                                dout <= SM_RB_DATA_RDY_o;
                        default:
                                dout <= 16'b0;
                        endcase
                end
        end
end


selectmap_readback selectmap_readback0(.T_CLOCK_i(clk),

        .CLOCK_i(clk),

        .RESET_i(rst),

        .T_CCLK_o(T_CCLK_o),

        .T_SELECTMAP_FLASH_BASE_i({flash_addr_high,flash_addr_low}),

        .T_SELECTMAP_INIT_o(T_SELECTMAP_INIT_o),

        .T_SELECTMAP_WRITE_o(T_SELECTMAP_WRITE_o),

        .T_SELECTMAP_CS_o(T_SELECTMAP_CS_o),

        .T_SELECTMAP_DATA_o(T_SELECTMAP_DATA_o),

        .T_SELECTMAP_DATA_i(T_SELECTMAP_DATA_i),

        .SM_RB_RQST_i(rb_req),

        .SM_RB_STATUS_o(SM_RB_STATUS_o),
```

```
            .SM_RB_ACK_i(SM_RB_ACK_i),

            .T_FLASH_DATA_i(T_FLASH_DATA_i),

            .T_FLASH_ADDRESS_o(T_FLASH_ADDRESS_o),

            .SM_RB_READ_DATA_o(SM_RB_READ_DATA_o),

            .SM_RB_ERROR_LOC_H_o(SM_RB_ERROR_LOC_H_o),

            .SM_RB_ERROR_LOC_L_o(SM_RB_ERROR_LOC_L_o),

            .SM_RB_ERROR_WORD_o(SM_RB_ERROR_WORD_o),

            .SM_RB_DATA_RDY_o(SM_RB_DATA_RDY_o));



endmodule



module selectmap_rb_xsoc(clk, rst, ctrl, int_req, sel, d, errorbus,
                          T_SELECTMAP_INIT_o, T_SELECTMAP_WRITE_o,
                          T_SELECTMAP_CS_o, T_SELECTMAP_DATA_o,
                          T_SELECTMAP_DATA_i,
                          T_FLASH_DATA_i, T_FLASH_ADDRESS_o,
                          SM_RB_STATUS_o, T_CCLK_o);

   input                  clk;
   input                  rst;
   input [47:0]     ctrl;
      output [2:0]        int_req;
   input [2:0]      sel;
   inout [47:0]     d;
      inout [15:0] errorbus;
      output              T_SELECTMAP_INIT_o;
      output              T_SELECTMAP_WRITE_o;
      output              T_SELECTMAP_CS_o;
      output [7:0] T_SELECTMAP_DATA_o;
      input [7:0]         T_SELECTMAP_DATA_i;
      input [15:0] T_FLASH_DATA_i;
      output[21:0] T_FLASH_ADDRESS_o;
      output              SM_RB_STATUS_o;
      output              T_CCLK_o;

      wire   [2:0]        int_req;
      wire                       clk;         // Clock
      wire                       rst;         // Reset
      wire   [47:0] ctrl;        //     16-bit memory control bus
      wire   [2:0]        sel;          // 1-bit peripheral select # (IO Base
Select)
                                                          // I/O 0 is at base
address FF00, subsequent IDs
                                                          //     are each 20h
higher
      tri    [47:0] d;                 // 16-bit tri-state data bus
      wor    [15:0]       errorbus;//  16-bit Global Error Bus, wired-OR

      wire   [14:0] addr;        // 5-bit Peripheral Address Space
```

```
      wire   [2:0]        ud_t;        // Active-low upper byte data bus tri-
state drive enable
      wire   [2:0]        ld_t;        // Active-low lower byte data bus tri-
state drive enable
      wire   [2:0]        ud_ce; //  Active-high  upper  byte  valid  data  clock
enable
      wire   [2:0]        ld_ce; //  Active-high  lower  byte  valid  data  clock
enable
      wire   [15:0] v_d;          // voted data bus
      wire   [4:0]        v_addr;      // voted address
      wire                       v_ld_ce;
      wire                       v_ld_t;

   wire                       SM_RB_STATUS_o;
      wire [7:0]          SM_RB_READ_DATA_o;
      wire [7:0]   SM_RB_ERROR_LOC_H_o;
      wire [15:0]         SM_RB_ERROR_LOC_L_o;
      wire [15:0]         SM_RB_ERROR_WORD_o;
      wire                       SM_RB_DATA_RDY_o;

      reg                        rb_req;// configure request
      reg [15:0]          flash_addr_low;
      reg [4:0]           flash_addr_high;

      wire   [47:0] dout;         // 16-bit data bus

      wire   [2:0]        CCLK_o;
      wire   [2:0]        RB_STATUS;
      wire [2:0]          SELECTMAP_INIT_o;
      wire [2:0]          SELECTMAP_WRITE_o;
      wire [2:0]          SELECTMAP_CS_o;
      wire [23:0]         SELECTMAP_DATA_o;
      wire [65:0]         FLASH_ADDRESS_o;

      ctrl_dec_syn   dec0(.clk(clk),   .rst(rst),   .ctrl(ctrl),   .sel(sel),
.ud_t(ud_t[0]),
                          .ld_t(ld_t[0]),  .ud_ce(ud_ce[0]),  .ld_ce(ld_ce[0]),
.addr(addr[4:0]),
                          .errorbus(errorbus), .id(`SMR_CTRL_DEC));

      ctrl_dec_syn   dec1(.clk(clk),   .rst(rst),   .ctrl(ctrl),   .sel(sel),
.ud_t(ud_t[1]),
                          .ld_t(ld_t[1]),  .ud_ce(ud_ce[1]),  .ld_ce(ld_ce[1]),
.addr(addr[9:5]),
                          .errorbus(errorbus), .id(`SMR_CTRL_DEC));


      ctrl_dec_syn   dec2(.clk(clk),   .rst(rst),   .ctrl(ctrl),   .sel(sel),
.ud_t(ud_t[2]),
                          .ld_t(ld_t[2]),  .ud_ce(ud_ce[2]),  .ld_ce(ld_ce[2]),
.addr(addr[14:10]),
                          .errorbus(errorbus), .id(`SMR_CTRL_DEC));

      voter1nsynh #(1)    SELECTMAP_INIT_v             (.din(SELECTMAP_INIT_o),
.dout(T_SELECTMAP_INIT_o),
                  .errorbus(errorbus),          .clk(clk),          .rst(rst),
.ID(`SMR_SELECTMAP_INIT));
      voter1nsynh #(1)    SELECTMAP_WRITE_v           (.din(SELECTMAP_WRITE_o),
.dout(T_SELECTMAP_WRITE_o),
                  .errorbus(errorbus),          .clk(clk),          .rst(rst),
.ID(`SMR_SELECTMAP_WRITE));
```

```
	voter1nsynh #(1)    SELECTMAP_CS_v                  (.din(SELECTMAP_CS_o),
.dout(T_SELECTMAP_CS_o),
		.errorbus(errorbus),          .clk(clk),          .rst(rst),
.ID(`SMR_SELECTMAP_CS));
	voter1nsynh #(8)    SELECTMAP_DATA_v              (.din(SELECTMAP_DATA_o),
.dout(T_SELECTMAP_DATA_o),
		.errorbus(errorbus),          .clk(clk),          .rst(rst),
.ID(`SMR_SELECTMAP_DATA));
	voter1nsynh #(22)   FLASH_ADDRESS_v                (.din(FLASH_ADDRESS_o),
.dout(T_FLASH_ADDRESS_o),
		.errorbus(errorbus),          .clk(clk),          .rst(rst),
.ID(`SMR_FLASH_ADDR));
	voter1nsynh #(1)    RB_STATUS_v (.din(RB_STATUS), .dout(SM_RB_STATUS_o),
		.errorbus(errorbus),          .clk(clk),          .rst(rst),
.ID(`SMR_RB_STATUS));
	//voted clock prone to noise on transitions, bad idea
	//voter1nsynh #(1)  CCLK_v        (.din(CCLK_o),        .dout(T_CCLK_o),
.errorbus(errorbus),
	//          .clk(clk), .rst(rst), .ID(`SMR_CCLK));

	assign T_CCLK_o = CCLK_o[0];


	selectmap_rb_xsoc_mod selectmap_readback0(.clk(clk),

		.rst(rst),

		.ld_ce(ld_ce),

		.ld_t(ld_t),

		.addr(addr),

		.d(d),

		.dout(dout[15:0]),

		.errorbus(errorbus),

		.int_req(int_req[0]),

		.T_CCLK_o(CCLK_o[0]),

		.T_SELECTMAP_INIT_o(SELECTMAP_INIT_o[0]),

		.T_SELECTMAP_WRITE_o(SELECTMAP_WRITE_o[0]),

		.T_SELECTMAP_CS_o(SELECTMAP_CS_o[0]),

		.T_SELECTMAP_DATA_o(SELECTMAP_DATA_o[7:0]),

		.T_SELECTMAP_DATA_i(T_SELECTMAP_DATA_i),

		.SM_RB_STATUS_o(RB_STATUS[0]),

		.T_FLASH_DATA_i(T_FLASH_DATA_i),

		.T_FLASH_ADDRESS_o(FLASH_ADDRESS_o[21:0]));

	selectmap_rb_xsoc_mod selectmap_readback1(.clk(clk),
```

```
        .rst(rst),
        .ld_ce(ld_ce),
        .ld_t(ld_t),
        .addr(addr),
        .d(d),
        .dout(dout[31:16]),
        .errorbus(errorbus),
        .int_req(int_req[1]),
        .T_CCLK_o(CCLK_o[1]),
        .T_SELECTMAP_INIT_o(SELECTMAP_INIT_o[1]),
        .T_SELECTMAP_WRITE_o(SELECTMAP_WRITE_o[1]),
        .T_SELECTMAP_CS_o(SELECTMAP_CS_o[1]),
        .T_SELECTMAP_DATA_o(SELECTMAP_DATA_o[15:8]),
        .T_SELECTMAP_DATA_i(T_SELECTMAP_DATA_i),
        .SM_RB_STATUS_o(RB_STATUS[1]),
        .T_FLASH_DATA_i(T_FLASH_DATA_i),
        .T_FLASH_ADDRESS_o(FLASH_ADDRESS_o[43:22]));

selectmap_rb_xsoc_mod selectmap_readback2(.clk(clk),
        .rst(rst),
        .ld_ce(ld_ce),
        .ld_t(ld_t),
        .addr(addr),
        .d(d),
        .dout(dout[47:32]),
        .errorbus(errorbus),
        .int_req(int_req[2]),
        .T_CCLK_o(CCLK_o[2]),
        .T_SELECTMAP_INIT_o(SELECTMAP_INIT_o[2]),
        .T_SELECTMAP_WRITE_o(SELECTMAP_WRITE_o[2]),
        .T_SELECTMAP_CS_o(SELECTMAP_CS_o[2]),
```

```
        .T_SELECTMAP_DATA_o(SELECTMAP_DATA_o[23:16]),

        .T_SELECTMAP_DATA_i(T_SELECTMAP_DATA_i),

        .SM_RB_STATUS_o(RB_STATUS[2]),

        .T_FLASH_DATA_i(T_FLASH_DATA_i),

        .T_FLASH_ADDRESS_o(FLASH_ADDRESS_o[65:44]));

    assign d[7:0]            = ld_t[0] ? 8'bz : dout[7:0];
    assign d[15:8]           = ud_t[0] ? 8'bz : dout[15:8];
    assign d[23:16]     = ld_t[1] ? 8'bz : dout[23:16];
    assign d[31:24]     = ud_t[1] ? 8'bz : dout[31:24];
    assign d[39:32]     = ld_t[2] ? 8'bz : dout[39:32];
    assign d[47:40]     = ud_t[2] ? 8'bz : dout[47:40];

endmodule
```

Q. TMRND.V

```
`timescale 1ns / 1ps
// file contains a pulldown for the processor data bus
// and various voter modules
// voter inputs floating cause false error reports

module datapulldown(d);
inout [47:0] d;
tri [47:0] d;

    genvar i;

    generate
    for (i=0;i<48;i=i+1) begin :pull_1
        PULLDOWN data_pull(.O(d[i]));
    end
    endgenerate


endmodule



// parametized voter, no error syndrome
module voter1n(din, dout);
    parameter W = 16;
    parameter N   = W-1; // msb index
    parameter N1 = (W*3)-1;

    input [N1:0] din;
    output [N:0] dout;
    reg [N:0] dout;

    genvar i;

    generate
    for (i=0;i<W;i=i+1) begin :maj1
        always @(din)
```

```
                begin
                        dout[i] = (din[i]&din[i+W]) | (din[i]&din[i+(2*W)]) |
                                (din[i+W]&din[i+(2*W)]);
                end
        end
        endgenerate

endmodule

// uses the same ports as the error detection voters but doesn't implement
// used to easily turn on/off errorbus for saving space on the FPGA
module voter1nsynh(din, dout, errorbus, clk, rst, ID);
        parameter W = 16;
        parameter N  = W-1; // msb index
        parameter N1 = (W*3)-1;

        input [N1:0] din;
        input  clk;
        input  rst;
        input [9:0] ID;
        input [15:0] errorbus;
        output [N:0] dout;

        reg [N:0] dout;

        genvar i;

        generate
        for (i=0;i<W;i=i+1) begin :maj1
                always @(din)
                begin
                        dout[i] = (din[i]&din[i+W]) | (din[i]&din[i+(2*W)]) |
                                (din[i+W]&din[i+(2*W)]);
                end
        end
        endgenerate

endmodule

// slower parametized voter with syndrome info that uses less resources
module voter1nsyn(din, dout, errorbus, clk, rst, ID);
        parameter W = 16;
        parameter N  = W-1; // msb index
        parameter N1 = (W*3)-1;

        input [N1:0] din;
        input  clk;
        input  rst;
        input [9:0] ID;
        inout [15:0] errorbus;
        output [N:0] dout;

        wor [15:0] errorbus;

        wor aerror_nxt, berror_nxt, cerror_nxt;

        reg    aerror, berror, cerror;

        genvar i;

        generate
```

```
	assign aerror_nxt = 1'b0;
	assign berror_nxt = 1'b0;
	assign cerror_nxt = 1'b0;

	for (i=0;i<W;i=i+1) begin :maj1
		assign dout[i] = (din[i]&din[i+W]) | (din[i]&din[i+(2*W)]) |
(din[i+W]&din[i+(2*W)]);
		assign aerror_nxt = ((din[i] & (~din[i+W]) & (~din[i+(2*W)])) |
((~din[i]) & din[i+W] & din[i+(2*W)])) ? 1'b1 : 1'b0;
		assign berror_nxt = (((~din[i]) & din[i+W] & (~din[i+(2*W)])) |
(din[i] & (~din[i+W]) & din[i+(2*W)])) ? 1'b1 : 1'b0;
		assign cerror_nxt = (((~din[i]) & (~din[i+W]) & din[i+(2*W)]) |
(din[i] & din[i+W] & (~din[i+(2*W)]))) ? 1'b1 : 1'b0;
	end
	endgenerate

	always @(posedge clk or posedge rst) begin
		if (rst) begin
			aerror <=0;
			berror <=0;
			cerror <=0;
		end
		else begin
			aerror <= aerror_nxt;
			berror <= berror_nxt;
			cerror <= cerror_nxt;
		end
	end

	// I have an error, drive error bus with my ID & info
	assign errorbus[14:5] = (aerror | berror | cerror) ? ID : 10'd0;
	assign errorbus[4:3] = 2'd0;
	assign errorbus[2:0] = (aerror | berror | cerror) ?
{cerror,berror,aerror} : 3'd0;

	// collision detect

	assign errorbus[15] = (aerror | berror | cerror) ?
((errorbus[14:0]==({ID,2'd0,cerror,berror,aerror})) ? 1'b0 : 1'b1) : 1'b0;


endmodule

// faster parametized voter that uses more resources
module voter1nsyn_new(din, dout, errorbus, clk, rst, ID);
	parameter W = 16;
	parameter N  = W-1; // msb index
	parameter N1 = (W*3)-1;

	input [N1:0] din;
	input  clk;
	input  rst;
	input [9:0] ID;
	inout [15:0] errorbus;
	output [N:0] dout;

	wor [15:0] errorbus;

	reg [N:0]    aerror;
	reg [N:0]    berror;
	reg [N:0]    cerror;
```

```
genvar i;

generate

for (i=0;i<W;i=i+1) begin :maj1
always @(posedge clk or posedge rst) begin
        if (rst) begin
                aerror[i] <=0;
                berror[i] <=0;
                cerror[i] <=0;
        end
        else begin
                aerror[i] <= ((din[i] & (~din[i+W]) & (~din[i+(2*W)])) |
((~din[i]) & din[i+W] & din[i+(2*W)]));
                berror[i] <= (((~din[i]) & din[i+W] & (~din[i+(2*W)])) |
(din[i] & (~din[i+W]) & din[i+(2*W)]));
                cerror[i] <= (((~din[i]) & (~din[i+W]) & din[i+(2*W)]) |
(din[i] & din[i+W] & (~din[i+(2*W)])));
        end
end

assign dout[i] = (din[i]&din[i+W]) | (din[i]&din[i+(2*W)]) |
(din[i+W]&din[i+(2*W)]);

end
endgenerate

// I have an error, drive error bus with my ID & info
assign errorbus[14:5] = ((|aerror) | (|berror) | (|cerror)) ? ID : 10'd0;
assign errorbus[4:3] = 2'd0;
assign errorbus[2:0] = ((|aerror) | (|berror) | (|cerror)) ?
{(|cerror),(|berror),(|aerror)} : 3'd0;

// collision detect

assign errorbus[15] = ((|aerror) | (|berror) | (|cerror)) ?
((errorbus[14:0]==({ID,2'd0,(|cerror),(|berror),(|aerror)})) ? 1'b0 : 1'b1) :
1'b0;


endmodule
```

R. VOTERIDS.V

```
// voter ID definitions

//datapath

`define     DP_RF_WE_V        10'd1
`define     DP_RNA_V                10'd2
`define     DP_RNB_V                10'd3
`define     DP_RDEST_V        10'd4
`define     DP_FWD_V                10'd5
`define     DP_IMM_V                10'd6
`define     DP_SEXTIMM4_V 10'd7
`define     DP_ZEXTIMM4_V 10'd8
`define     DP_WORDIMM4_V 10'd9
`define     DP_IMM12_V        10'd10
```

```
`define	DP_PIPE_CE_V	10'd11
`define	DP_B15_4_CE_V	10'd12
`define	DP_ADD_V		10'd13
`define	DP_CI_V		10'd14
`define	DP_LOGICOP_V	10'd15
`define	DP_SRI_V		10'd16
`define	DP_SUM_T_V	10'd17
`define	DP_LOGIC_T_V	10'd18
`define	DP_SHL_T_V	10'd19
`define	DP_SHR_T_V	10'd20
`define	DP_ZEROEXT_T_V	10'd21
`define	DP_RET_T_V	10'd22
`define	DP_UD_T_V	10'd23
`define	DP_LD_T_V	10'd24
`define	DP_UDLT_T_V	10'd25
`define	DP_BRANCH_V	10'd26
`define	DP_BRDISP_V	10'd27
`define	DP_SELPC_V	10'd28
`define	DP_ZEROPC_V	10'd29
`define	DP_DMAPC_V	10'd30
`define	DP_PC_CE_V	10'd31
`define	DP_RET_CE_V	10'd32
`define	DP_RES_V		10'd33
`define	DP_A_V		10'd34
`define	DP_B_V		10'd35
`define	DP_DOUT_V	10'd36
`define	DP_RET_V		10'd37
`define	DP_ADDR_NXT_V	10'd38

`define	CO_RDY_V		10'd64
`define	CO_INT_REQ_V	10'd65
`define	CO_DMA_REQ_V	10'd66
`define	CO_ZERODMA_V	10'd67
`define	CO_A15_V		10'd68
`define	CO_Z_V		10'd69
`define	CO_N_V		10'd70
`define	CO_CO_V		10'd71
`define	CO_V__V		10'd72
`define	CO_INSN_V	10'd73
`define	CO_ADD_V		10'd74
`define	CO_CI_V		10'd75
`define	CO_BRANCH_V	10'd76
`define	CO_SUM_T_V	10'd77
`define	CO_LOGIC_T_V	10'd78
`define	CO_SHL_T_V	10'd79
`define	CO_SHR_T_V	10'd80
`define	CO_ZEROEXT_T_V	10'd81
`define	CO_RET_T_V	10'd82
`define	CO_IF_IR_V	10'd83
`define	CO_IR_V		10'd84
`define	CO_EX_IR_V	10'd85
`define	CO_EX_CALL_V	10'd86
`define	CO_EX_ST_V	10'd87
`define	CO_IFETCH_V	10'd88
`define	CO_DMA_V		10'd89
`define	CO_SYNC_RESET_V	10'd90
`define	CO_DC_ANNUL_V	10'd91
`define	CO_EX_ANNUL_V	10'd92
`define	CO_INT_PEND_V	10'd93
`define	CO_DC_INT_V	10'd94
`define	CO_DMA_PEND_V	10'd95
`define	CO_ZERO_PEND_V	10'd96
```

```
`define  COUNT64_COUNT0  10'd128
`define  COUNT64_COUNT1  10'd129
`define  COUNT64_COUNT2  10'd130
`define  COUNT64_COUNT3  10'd131
`define  COUNT64_ADDR    10'd132
`define  COUNT64_DIN     10'd133
`define      COUNT64_CTRL_DEC 10'd134
`define      COUNT64_LD_T  10'd135
`define      COUNT64_LD_CE  10'd136

`define      ER_CTRL_DEC               10'd140
`define      ER_MODULE_V               10'd141
`define      ER_QU_ADD_V               10'd142
`define      ER_QU_REM_V               10'd143
`define      ER_MO_ADDR_V        10'd143
`define      ER_MO_LD_T_V        10'd143

`define      PC104_CTRL_DEC            10'd150
`define      PC104_MO_D_V        10'd151
`define      PC104_MO_LD_CE_V    10'd152
`define      PC104_MO_LD_T_V     10'd153
`define      PC104_MO_ADDR_V     10'd154

`define      BRAM_SYNDROME_ERR   10'd160
`define      BRAM_ADDR_NXT0_V    10'd161
`define      BRAM_ADDR_NXT1_V    10'd162
`define      BRAM_ADDR_NXT2_V    10'd163

`define      MC_MEM_CE_V               10'd170
`define      MC_WORD_NXT_V       10'd171
`define      MC_READ_NXT_V       10'd172
`define      MC_DBUS_NXT_V       10'd173
`define      MC_DMA_V                        10'd174
`define      MC_ADDR_NXT_V       10'd175
`define      MC_DMA_REQ_IN_V     10'd176
`define      MC_ZERO_REQ_IN_V    10'd177

`define      SMC_CTRL_DEC        10'd180

`define      SMC_SELECTMAP_INIT  10'd200
`define      SMC_SELECTMAP_WRITE 10'd201
`define      SMC_SELECTMAP_CS          10'd202
`define      SMC_SELECTMAP_DATA  10'd203
`define      SMC_FLASH_ADDR                  10'd204
`define      SMC_CONFIG_STATUS         10'd205
`define      SMC_D                                 10'd206
`define      SMC_ADDR                              10'd207
`define      SMC_LD_T                              10'd208
`define      SMC_LD_CE                       10'd209

`define      SMR_CTRL_DEC              10'd190
`define      SMR_D                                 10'd220
`define      SMR_ADDR                              10'd221
`define      SMR_LD_T                              10'd222
`define      SMR_LD_CE                       10'd223
`define      SMR_SELECTMAP_INIT  10'd224
`define      SMR_SELECTMAP_WRITE 10'd225
`define      SMR_SELECTMAP_CS          10'd226
`define      SMR_SELECTMAP_DATA  10'd227
`define      SMR_FLASH_ADDR                  10'd228
`define      SMR_RB_STATUS             10'd229
```

```
`define      SMR_CCLK                                10'd230
```

S. X1CONTROL.V

```
`timescale 1ns / 1ps
`define FLASH_RP_LOW            16
`define X2_PROG_LOW             16

module x1control(T_VPPEN_o,
                                T_PROM_ENABLE_o,
                                DATA_FROM_X2_MULTCHK_i,
                                DATA_TO_X2_RESET_o,
                                DATA_FROM_X2_READY_i,
                                DATA_FROM_X2_OUTPUT_i,
                                T_FLASH_DATA_i,
                                T_FLASH_ADDRESS_o,
                                T_FLASH_WE_o,
                                T_FLASH_RP_o,
                                T_FLASH_WP_o,
                                T_FLASH_CE_A_o,
                                T_FLASH_OE_o,
                                T_DATA_io,
                                T_ADDRESS_i,
                                T_IOREAD_i,
                                T_IOWRITE_i,
                                T_IOCS_i,
                                T_INTRPT_o,
                                T_CCLK_o,
                                T_SELECTMAP_INIT_o,
                                T_SELECTMAP_WRITE_o,
                                T_SELECTMAP_CS_o,
                                T_SELECTMAP_DATA_io,
                                T_clock_i,
                                T_X2_MODE,
                                T_X2_PROG_o);
//                              v_addr_nxt,
//                              v_xd,
//                              int_req,
//                              int_test);


      output                    T_VPPEN_o; // # only needed for writing FLASH
      output                    T_PROM_ENABLE_o; // drive high to save power
on EEPROM
      output                    DATA_TO_X2_RESET_o; // X1 reset to X2
      input             DATA_FROM_X2_READY_i; // data ready on X2
      input [31:0] DATA_FROM_X2_OUTPUT_i; // Data from X2 experiment
      input [1:0]       DATA_FROM_X2_MULTCHK_i;
      // Flash Memory Pins
      input [15:0] T_FLASH_DATA_i;  // flash data in
      output [20:0]     T_FLASH_ADDRESS_o;  //flash address output
      output            T_FLASH_WE_o;  // flash write enable
      output                    T_FLASH_RP_o;
      output                    T_FLASH_WP_o;
      output                    T_FLASH_CE_A_o;
      output                    T_FLASH_OE_o;
      // PC-104 Interface Pins
      inout [7:0]  T_DATA_io;
```

```
	input [9:0]  T_ADDRESS_i;
	input               T_IOREAD_i;
	input               T_IOWRITE_i;
	input               T_IOCS_i;
	output                     T_INTRPT_o;
	// SelectMap Interface Pins
	output                     T_CCLK_o;      //   #Drive   X2's   CCLK   pin,
flight_board = p65
	output                     T_SELECTMAP_INIT_o;   //#Drive  X2's  INIT  pin,
flight_board = p124
	output                     T_SELECTMAP_WRITE_o;  //#Drive X2's WRITE pin,
flight_board = p63
	output                     T_SELECTMAP_CS_o;  // #Drive X2's CS pin
	inout [7:0]         T_SELECTMAP_DATA_io; // SelectMap data xfer
	// System clock
	input               T_clock_i; // 51Mhz SYSCLK from PC-104
	// ?? T_X2_MODE;
	output                     T_X2_PROG_o;
	output [2:0]  T_X2_MODE;

//	output              fifo_full;
//	output [15:0] dout;
//	output              load_status;
//	output [47:0] ctrl;
//	output [47:0] sel;

//	output [15:0] v_addr_nxt;
//	o//utput      [47:0]       areg,breg;
//	ou/tput       [47:0] memdta, xd;
//	output [15:0] v_xd;
//	output [47:0] d;
//	output [2:0]        int_req;
//	input                             int_test;

	//input                           rst;
	tri [7:0]           T_SELECTMAP_DATA_io;

	// XSOC interface signals
	wire   [47:0] ctrl;
	wire   [2:0]        ctrl0;
	wire   [2:0]        int_req;
	wire   [23:0] io_int_req;
	wire [23:0]         sel;
	tri    [47:0] d;
	wor    [15:0] errorbus;

	// Global clock & reset
	wire                       clk;
	reg                        rst=1'b1;

	wire                       config_active;
	wire                       readback_active;
	wire [21:0]         flash_address_config;
	wire [21:0]         flash_address_rb;

	reg                        T_X2_PROG_o;
	reg                        T_FLASH_RP_o;
	reg                        flash_rp_cnt;
	reg                        x2_prog_cnt;

	wire [7:0]          s_selectmap_data_i;
	wire                       s_selectmap_WRITE_o;
```

```
      wire                        WRITE_config;
      wire                        INIT_config;
      wire                        CS_config;
      wire                        WRITE_readback;
      wire                        INIT_readback;
      wire                        CS_readback;
      wire                        CCLK_readback;
      wire [7:0]          selectmap_write_data_readback;
      wire [7:0]          selectmap_write_data_config;
      wire [7:0]          selectmap_write_data;
      wire                        dlbclk;


      wire   [15:0] v_addr_nxt, v_xd;
      wire   [47:0] addr_nxt, xd;

      wire                        T_IOREAD_i_out;


//    voter1n #(16) addr_nxt_v (.din(addr_nxt), .dout(v_addr_nxt));
//    voter1n #(16) xd_v (.din(xd), .dout(v_xd));


      assign T_VPPEN_o = 1'b0;
      assign T_PROM_ENABLE_o = 1'b1;
      assign T_X2_MODE = {1'b1, 1'b1, 1'b0};
      assign T_FLASH_OE_o = 1'b0;
      assign T_FLASH_WP_o = 1'b1;
      assign T_FLASH_WE_o = 1'b1;

      // turn off reset after 1 clock;
      always @(posedge clk) begin
             if (rst) begin
                    rst <= 1'b0;
             end
      end

      // stop ISE from complaining about T_READ_i
      IBUF read_i_buf (.I(T_IOREAD_i), .O(T_IOREAD_i_out));

      //~20Mhz system clock (51/2.5)
      clock_div clock_div1(.ACLK(T_clock_i),
                                                    .CLKDIV(clk),
                                                    .RST(1'b0));

//    clock_dbl clock_dbl1(.ACLK(clk), .CLKDBL(dblclk), .RST(1'b0));

      //xsoc microcontroller
      // ctrl0 is used to indicate additional wait states for IO
      // 0 - not ready, 1 - ready
      // comment out this line if using ctrl0
      assign ctrl0 = {1'b1, 1'b1, 1'b1};

      xsoc xsoc1( .clk(clk),
                          .rst(rst),
                          .ctrl(ctrl),
                          .ctrl0(ctrl0),
                          .int_req(int_req),
                   //     .int_req(3'h0),
                          .sel(sel),
                          .d(d),
                          .errorbus(errorbus),
```

```
                          .addr_nxt(addr_nxt));

//PC-104 Interface, ID #0
xsoc_pc104 xsoc_pc104_1    (.clk(clk),
                                                   .rst(rst),
                                                   .ctrl(ctrl),

.int_req({io_int_req[16],io_int_req[8],io_int_req[0]}),

.sel({sel[16],sel[8],sel[0]}),
                                                   .d(d),

.T_DATA_io(T_DATA_io),

.T_ADDRESS_i(T_ADDRESS_i),

.T_IOREAD_i(T_IOREAD_i_out),

.T_IOWRITE_i(T_IOWRITE_i),
                                                   .T_IOCS_i(T_IOCS_i),

.T_INTRPT_o(T_INTRPT_o),

.fifo_full(fifo_full),

.errorbus(errorbus));

// 64-bit counter, ID #3

//assign io_int_req[1]=int_test; // irq 1
//assign io_int_req[9]=int_test;
//assign io_int_req[17]=int_test;


// must turn off unused interrupts
//assign io_int_req[0]=1'b0; // irq 0
//assign io_int_req[8]=1'b0;
//assign io_int_req[16]=1'b0;
//assign io_int_req[1]=1'b0; // irq 1
//assign io_int_req[9]=1'b0;
//assign io_int_req[17]=1'b0;
assign io_int_req[2]=1'b0; // irq 2
assign io_int_req[10]=1'b0;
assign io_int_req[18]=1'b0;
assign io_int_req[3]=1'b0;  // irq 3
assign io_int_req[11]=1'b0;
assign io_int_req[19]=1'b0;
assign io_int_req[4]=1'b0;       // irq 4
assign io_int_req[12]=1'b0;
assign io_int_req[20]=1'b0;
//assign io_int_req[5]=1'b0;            // irq 5
//assign io_int_req[13]=1'b0;
//assign io_int_req[21]=1'b0;
assign io_int_req[6]=1'b0;       // irq 6
assign io_int_req[14]=1'b0;
assign io_int_req[22]=1'b0;
assign io_int_req[7]=1'b0;       // irq 7
assign io_int_req[15]=1'b0;
assign io_int_req[23]=1'b0;

xscounter64 xscounter64_1( .clk(clk),
                                                         .rst(rst),
```

```
                                                                .ctrl(ctrl),

       .int_req_in(io_int_req),

       .int_req_out(int_req),

       .sel({sel[19],sel[11], sel[3]}),
                                                                .d(d),

       .errorbus(errorbus));

       // Error Reporting Module, ID # 1
       errorrept errorrept1 (           .clk(clk),
                                                                .rst(rst),
                                                                .ctrl(ctrl),

       .int_req({io_int_req[17],io_int_req[9],io_int_req[1]}),

       .sel({sel[17],sel[9],sel[1]}),
                                                                .d(d),

       .errorbus(errorbus));


       assign T_FLASH_ADDRESS_o = config_active ? flash_address_config[20:0] :
flash_address_rb[20:0];
       assign  T_FLASH_CE_A_o  =  config_active  ?  flash_address_config[21]  :
(readback_active ? flash_address_rb[21] : 1'b1);

       // delay on X2_PROG by X2_PROG_LOW CLOCKS
       always @(posedge clk or posedge rst) begin
              if (rst) begin
                     x2_prog_cnt <= 0;
                     T_X2_PROG_o <= 0;
              end
              else begin
                     if (x2_prog_cnt == `X2_PROG_LOW)
                            begin
                                   T_X2_PROG_o <= 1;
                                   x2_prog_cnt <= `X2_PROG_LOW;
                            end
                     else
                            begin
                                   x2_prog_cnt <= x2_prog_cnt + 1;
                            end
              end
       end

       // delay on T_FLASH_RP by FLASH_RP_LOW clocks
       always @(posedge clk or posedge rst) begin
              if (rst) begin
             T_FLASH_RP_o <= 0;
             flash_rp_cnt <= 0;
              end
              else begin
                     if (flash_rp_cnt == `FLASH_RP_LOW)
                            begin
                 T_FLASH_RP_o <= 1;
                 flash_rp_cnt <= `FLASH_RP_LOW;
                            end
                     else
                            begin
```

```
                    flash_rp_cnt <= flash_rp_cnt + 1;
                end
            end
        end

        assign        selectmap_write_data        =        readback_active        ?
selectmap_write_data_readback : selectmap_write_data_config;
        assign   s_selectmap_WRITE_o   =   readback_active   ?   WRITE_readback   :
WRITE_config;
        assign s_selectmap_data_i = T_SELECTMAP_DATA_io;
        assign T_SELECTMAP_DATA_io = ~s_selectmap_WRITE_o ? selectmap_write_data
: 8'bz;
        assign T_SELECTMAP_WRITE_o = s_selectmap_WRITE_o;
        assign   T_SELECTMAP_INIT_o     =   readback_active   ?   INIT_readback   :
INIT_config;
        assign T_SELECTMAP_CS_o = readback_active ? CS_readback : CS_config;
        assign T_CCLK_o = CCLK_readback;

        // SelectMap config, ID # 4
        selectmap_config_xsoc selectmap_config_xsoc0(.clk(clk),

                    .rst(rst),

                    .ctrl(ctrl),

                    .sel({sel[20],sel[12],sel[4]}),

                    .d(d),

                    .errorbus(errorbus),

                    .T_SELECTMAP_INIT_o(INIT_config),

                    .T_SELECTMAP_WRITE_o(WRITE_config),

                    .T_SELECTMAP_CS_o(CS_config),

                    .T_SELECTMAP_DATA_o(selectmap_write_data_config),

                    .T_FLASH_DATA_i(T_FLASH_DATA_i),

                    .T_FLASH_ADDRESS_o(flash_address_config),

                    .SM_CONFIG_STATUS_o(config_active));

        selectmap_rb_xsoc selectmap_rb_xsoc0(                .clk(clk),

                    .rst(rst),

                    .ctrl(ctrl),


        .int_req({io_int_req[21],io_int_req[13],io_int_req[5]}),

                    .sel({sel[21],sel[13],sel[5]}),

                    .d(d),

                    .errorbus(errorbus),

                    .T_SELECTMAP_INIT_o(INIT_readback),
```

```
                            .T_SELECTMAP_WRITE_o(WRITE_readback),

                            .T_SELECTMAP_CS_o(CS_readback),

                            .T_SELECTMAP_DATA_o(selectmap_write_data_readback),

                            .T_SELECTMAP_DATA_i(s_selectmap_data_i),

                            .T_FLASH_DATA_i(T_FLASH_DATA_i),

                            .T_FLASH_ADDRESS_o(flash_address_rb),

                            .SM_RB_STATUS_o(readback_active),

                            .T_CCLK_o(CCLK_readback));

      // no X2 interface yet
      assign DATA_TO_X2_RESET_o = rst;


endmodule
```

T. XR16.V

```
/* xr16.v -- xr16 pipelined RISC processor synthesizable Verilog model
 *
 * Copyright (C) 1999, 2000, Gray Research LLC.  All rights reserved.
 * The contents of this file are subject to the XSOC License Agreement;
 * you may not use this file except in compliance with this Agreement.
 * See the LICENSE file.
 *
 * $Header: /dist/xsocv/xr16.v 7     4/06/00 10:55a Jan $
 * $Log: /dist/xsocv/xr16.v $
 *
 * 7     4/06/00 10:55a Jan
 * polish
 */
 /* last modified by David Dwiggins, 24 August 2008.  Adapted to use
 *  TMR/ECC functionality for fault-tolerance.  */
 // original code by Jan Gray included at end of file, commented out
`timescale 1ns / 1ps

module xr16(clk, rst, rdy, ud_t, ld_t, udlt_t, int_req, dma_req, zerodma,
    insn, mem_ce, word_nxt, read_nxt, dbus_nxt, dma, addr_nxt, d, areg,
breg,pipe_ce,memdta,errorbus,
        if_ir, ir, ex_ir, fwd, imm, sextimm4, zextimm4, wordimm4, imm12,
drivelo, drivehi);

      parameter           W      = 16;  // word width
      parameter           N      = W-1; // msb index

      input               clk;          // global clock
      input               rst;          // global async reset
      input  [2:0]  rdy;          // current memory access is ready
      input  [2:0]  ud_t;         // active low store data MSB output enable
      input  [2:0]  ld_t;         // active low store data LSB output enable
```

```
	input  [2:0]  udlt_t;             // active low store data MSB->LSB output
en
	input  [2:0]  int_req;      // interrupt request
	input  [2:0]  dma_req;      // DMA request
	input  [2:0]  zerodma;      // zero DMA counter request
	input  [47:0] insn;         // new instruction word
	inout  [15:0] errorbus;
	output [2:0]         mem_ce;              // memory access clock enable
	output [2:0]         word_nxt;     // next access is word wide
	output [2:0]         read_nxt;     // next access is read
	output [2:0]         dbus_nxt;     // next access uses on-chip data bus
	output [2:0]         dma;          // current access is a DMA transfer
	output [47:0] addr_nxt;     // address of next memory access
	inout  [47:0] d;                   // on-chip data bus
	output [47:0] areg;
	output [47:0] breg;
	output [2:0] pipe_ce;
	output [47:0] memdta;
	output [47:0] if_ir;
	output [47:0] ir;
	output [47:0] ex_ir;
	output [2:0] fwd;
	output [35:0]  imm;
	output [2:0] sextimm4;
	output [2:0] zextimm4;
	output [2:0] wordimm4;
	output [2:0] imm12;
	output [2:0] drivelo;
	output [2:0] drivehi;

	tri    [47:0] d;
	wire   [2:0]         int_req;

	// locals
	wire   [2:0]  a15;          // A operand msb
	wire   [2:0]  z;                   // zero result condition code
	wire   [2:0]  n;                   // negative result condition code
	wire   [2:0]  co;                  // carry-out result codition code
	wire   [2:0]  v;                   // oVerflow result condition code
	wire   [2:0]  rf_we;        // register file write enable
	wire [11:0]   rna;          // register file port A register number
	wire [11:0]   rnb;          // register file port B register number
	wire [11:0]      rdest;    // register file write port register number
	wire   [2:0]  fwd;          // forward result bus into A operand register
	wire [35:0]   imm;          // 12-bit immediate field
	wire   [2:0]  sextimm4;     // sign-extend 4-bit immediate operand
	wire   [2:0]  zextimm4;     // zero-extend 4-bit immediate operand
	wire   [2:0]  wordimm4;     // word-offset 4-bit immediate operand
	wire   [2:0]  imm12;        // 12-bit immediate operand
	wire   [2:0]  pipe_ce;      // pipeline clock enable
	wire   [2:0]  b15_4_ce;     // b[15:4] clock enable
	wire   [2:0]  add;          // 1 => A + B; 0 => A - B
	wire   [2:0]  ci;                  // carry-in
	wire [5:0]    logicop;      // logic unit opcode
	wire   [2:0]  sri;          // shift right msb input
	wire   [2:0]  sum_t;        // active low adder output enable
	wire   [2:0]  logic_t;      // active low logic unit output enable
	wire   [2:0]  zeroext_t;    // active low zero-extension output enable
	wire   [2:0]  shr_t;        // active low shift right output enable
	wire   [2:0]  shl_t;        // active low shift left output enable
	wire   [2:0]  ret_t;        // active low return address output enable
	wire   [2:0]  branch;              // branch taken
```

```
	wire [23:0]  brdisp;		// 8-bit branch displacement
	wire   [2:0]  selpc;	// address mux selects next PC
	wire   [2:0]  zeropc;		// force next PC to 0
	wire   [2:0]  dmapc;	// use DMA register in PC register file
	wire   [2:0]  pc_ce;	// PC clock enable
	wire   [2:0]  ret_ce;		// return address clock enable
	wire  [47:0] areg, breg;
	wire  [47:0] memdta;
	wire   [47:0] a;
	wire   [47:0] b;
	wire   [47:0] dout;
	wire   [47:0] ret;

	wor    [15:0] errorbus;

	// control voter wires

	wire   [2:0]  c_add, c_ci, c_branch, c_sum_t, c_logic_t, c_shl_t, c_shr_t,
c_zeroext_t, c_ret_t;
	wire   [47:0] c_if_ir, c_ir, c_ex_ir;
	wire   [2:0]  c_ex_call,   c_ex_st,   c_ifetch,   c_dma,   c_sync_reset,
c_dc_annul;
	wire  [2:0] c_ex_annul, c_int_pend, c_dc_int, c_dma_pend, c_zero_pend;

	wire   [2:0]  drivehi;
	wire   [2:0]  drivelo;
	wire   [47:0] my_res;

	// submodules

//due to partitioning not supporting partitions called from generate loops
//generate loops expanded by hand

	datapath dp1(
		.clk(clk), .rst(rst),
		.rf_we(rf_we), .rna(rna), .rnb(rnb), .rdest(rdest),
		.fwd(fwd[0]),       .imm(imm[11:0]),       .sextimm4(sextimm4[0]),
.zextimm4(zextimm4[0]),
		.wordimm4(wordimm4[0]), .imm12(imm12[0]),
		.pipe_ce(pipe_ce), .b15_4_ce(b15_4_ce),
		.add(add[0]), .ci(ci[0]), .logicop(logicop[1:0]), .sri(sri[0]),
		.sum_t(sum_t[0]),      .logic_t(logic_t[0]),      .shl_t(shl_t[0]),
.shr_t(shr_t[0]),
		.zeroext_t(zeroext_t[0]), .ret_t(ret_t[0]),
		.ld_t(ld_t[0]), .ud_t(ud_t[0]), .udlt_t(udlt_t[0]),
		.branch(branch[0]),     .brdisp(brdisp[7:0]),     .selpc(selpc[0]),
.zeropc(zeropc[0]),
		.dmapc(dmapc), .pc_ce(pc_ce), .ret_ce(ret_ce),
		.a15(a15[0]), .z(z[0]), .n(n[0]), .co(co[0]), .v(v[0]),
		.addr_nxt(addr_nxt[15:0]), .res(d), .areg(areg[15:0]),
		.breg(breg[15:0]), .memdta(memdta[15:0]),
		.my_res(my_res[15:0]),
		.in_a(a),.out_a(a[15:0]),
		.in_b(b),.out_b(b[15:0]),
		.in_dout(dout),.out_dout(dout[15:0]),
		.in_ret(ret),.out_ret(ret[15:0]),
		.errorbus(errorbus),           .inst_id(2'd1),.drivehi(drivehi[0]),
.drivelo(drivelo[0]),
		.in_addr_nxt(addr_nxt));

		datapath dp2(
		.clk(clk), .rst(rst),
```

```
		.rf_we(rf_we), .rna(rna), .rnb(rnb), .rdest(rdest),
		.fwd(fwd[1]),	.imm(imm[23:12]),	.sextimm4(sextimm4[1]),
.zextimm4(zextimm4[1]),
		.wordimm4(wordimm4[1]), .imm12(imm12[1]),
		.pipe_ce(pipe_ce), .b15_4_ce(b15_4_ce),
		.add(add[1]), .ci(ci[1]), .logicop(logicop[3:2]), .sri(sri[1]),
		.sum_t(sum_t[1]),	.logic_t(logic_t[1]),	.shl_t(shl_t[1]),
.shr_t(shr_t[1]),
		.zeroext_t(zeroext_t[1]), .ret_t(ret_t[1]),
		.ld_t(ld_t[1]), .ud_t(ud_t[1]), .udlt_t(udlt_t[1]),
		.branch(branch[1]),	.brdisp(brdisp[15:8]),	.selpc(selpc[1]),
.zeropc(zeropc[1]),
		.dmapc(dmapc), .pc_ce(pc_ce), .ret_ce(ret_ce),
		.a15(a15[1]), .z(z[1]), .n(n[1]), .co(co[1]), .v(v[1]),
		.addr_nxt(addr_nxt[31:16]), .res(d), .areg(areg[31:16]),
		.breg(breg[31:16]), .memdta(memdta[31:16]),
		.my_res(my_res[31:16]),
		.in_a(a),.out_a(a[31:16]),
		.in_b(b),.out_b(b[31:16]),
		.in_dout(dout),.out_dout(dout[31:16]),
		.in_ret(ret),.out_ret(ret[31:16]),
		.errorbus(errorbus),	.inst_id(2'd2),	.drivehi(drivehi[1]),
.drivelo(drivelo[1]),
		.in_addr_nxt(addr_nxt));

		datapath dp3(
		.clk(clk), .rst(rst),
		.rf_we(rf_we), .rna(rna), .rnb(rnb), .rdest(rdest),
		.fwd(fwd[2]),	.imm(imm[35:24]),	.sextimm4(sextimm4[2]),
.zextimm4(zextimm4[2]),
		.wordimm4(wordimm4[2]), .imm12(imm12[2]),
		.pipe_ce(pipe_ce), .b15_4_ce(b15_4_ce),
		.add(add[2]), .ci(ci[2]), .logicop(logicop[5:4]), .sri(sri[2]),
		.sum_t(sum_t[2]),	.logic_t(logic_t[2]),	.shl_t(shl_t[2]),
.shr_t(shr_t[2]),
		.zeroext_t(zeroext_t[2]), .ret_t(ret_t[2]),
		.ld_t(ld_t[2]), .ud_t(ud_t[2]), .udlt_t(udlt_t[2]),
		.branch(branch[2]),	.brdisp(brdisp[23:16]),	.selpc(selpc[2]),
.zeropc(zeropc[2]),
		.dmapc(dmapc), .pc_ce(pc_ce), .ret_ce(ret_ce),
		.a15(a15[2]), .z(z[2]), .n(n[2]), .co(co[2]), .v(v[2]),
		.addr_nxt(addr_nxt[47:32]), .res(d), .areg(areg[47:32]),
		.breg(breg[47:32]), .memdta(memdta[47:32]),
		.my_res(my_res[47:32]),
		.in_a(a),.out_a(a[47:32]),
		.in_b(b),.out_b(b[47:32]),
		.in_dout(dout),.out_dout(dout[47:32]),
		.in_ret(ret),.out_ret(ret[47:32]),
		.errorbus(errorbus),	.inst_id(2'd3),	.drivehi(drivehi[2]),
.drivelo(drivelo[2]),
		.in_addr_nxt(addr_nxt));



		control ctrl1(
		.clk(clk), .rst(rst), .rdy(rdy),
		.int_req(int_req[0]), .dma_req(dma_req[0]), .zerodma(zerodma[0]),
		.insn(insn[15:0]), .a15(a15[0]), .z(z[0]), .n(n[0]), .co(co[0]),
.v(v[0]),
		.mem_ce(mem_ce[0]),			.word_nxt(word_nxt[0]),
.read_nxt(read_nxt[0]),
		.dbus_nxt(dbus_nxt[0]), .dma(dma[0]),
```

```
            .rf_we(rf_we[0]),         .rna(rna[3:0]),         .rnb(rnb[3:0]),
.rdest(rdest[3:0]),
            .fwd(fwd[0]),       .imm(imm[11:0]),       .sextimm4(sextimm4[0]),
.zextimm4(zextimm4[0]),
            .wordimm4(wordimm4[0]), .imm12(imm12[0]),
            .pipe_ce(pipe_ce[0]), .b15_4_ce(b15_4_ce[0]),
            .add(add[0]), .ci(ci[0]), .logicop(logicop[1:0]), .sri(sri[0]),
            .sum_t(sum_t[0]),     .logic_t(logic_t[0]),     .shl_t(shl_t[0]),
.shr_t(shr_t[0]),
            .zeroext_t(zeroext_t[0]), .ret_t(ret_t[0]),
            .branch(branch[0]),    .brdisp(brdisp[7:0]),    .selpc(selpc[0]),
.zeropc(zeropc[0]),
            .dmapc(dmapc[0]), .pc_ce(pc_ce[0]), .ret_ce(ret_ce[0]),
            .in_add(c_add),         .out_add(c_add[0]),         .in_ci(c_ci),
.out_ci(c_ci[0]), .in_branch(c_branch), .out_branch(c_branch[0]),
            .in_sum_t(c_sum_t),                          .out_sum_t(c_sum_t[0]),
.in_logic_t(c_logic_t), .out_logic_t(c_logic_t[0]),
            .in_shl_t(c_shl_t),   .out_shl_t(c_shl_t[0]),   .in_shr_t(c_shr_t),
.out_shr_t(c_shr_t[0]),
            .in_zeroext_t(c_zeroext_t),         .out_zeroext_t(c_zeroext_t[0]),
.in_ret_t(c_ret_t), .out_ret_t(c_ret_t[0]),
            .in_if_ir(c_if_ir), .out_if_ir(c_if_ir[15:0]),
            .in_ir(c_ir),       .out_ir(c_ir[15:0]),       .in_ex_ir(c_ex_ir),
.out_ex_ir(c_ex_ir[15:0]),
            .in_ex_call(c_ex_call),                 .out_ex_call(c_ex_call[0]),
.in_ex_st(c_ex_st), .out_ex_st(c_ex_st[0]),
            .in_ifetch(c_ifetch),   .out_ifetch(c_ifetch[0]),   .in_dma(c_dma),
.out_dma(c_dma[0]), .in_sync_reset(c_sync_reset),
            .out_sync_reset(c_sync_reset[0]),         .in_dc_annul(c_dc_annul),
.out_dc_annul(c_dc_annul[0]),
            .in_ex_annul(c_ex_annul),             .out_ex_annul(c_ex_annul[0]),
.in_int_pend(c_int_pend), .out_int_pend(c_int_pend[0]),
            .in_dc_int(c_dc_int),                       .out_dc_int(c_dc_int[0]),
.in_dma_pend(c_dma_pend), .out_dma_pend(c_dma_pend[0]),
            .in_zero_pend(c_zero_pend), .out_zero_pend(c_zero_pend[0]),
            .inum(2'd1),.errorbus(errorbus));

            control ctrl2(
            .clk(clk), .rst(rst), .rdy(rdy),
            .int_req(int_req[1]), .dma_req(dma_req[1]), .zerodma(zerodma[1]),
            .insn(insn[31:16]), .a15(a15[1]), .z(z[1]), .n(n[1]), .co(co[1]),
.v(v[1]),
            .mem_ce(mem_ce[1]),                         .word_nxt(word_nxt[1]),
.read_nxt(read_nxt[1]),
            .dbus_nxt(dbus_nxt[1]), .dma(dma[1]),
            .rf_we(rf_we[1]),         .rna(rna[7:4]),         .rnb(rnb[7:4]),
.rdest(rdest[7:4]),
            .fwd(fwd[1]),       .imm(imm[23:12]),       .sextimm4(sextimm4[1]),
.zextimm4(zextimm4[1]),
            .wordimm4(wordimm4[1]), .imm12(imm12[1]),
            .pipe_ce(pipe_ce[1]), .b15_4_ce(b15_4_ce[1]),
            .add(add[1]), .ci(ci[1]), .logicop(logicop[3:2]), .sri(sri[1]),
            .sum_t(sum_t[1]),     .logic_t(logic_t[1]),     .shl_t(shl_t[1]),
.shr_t(shr_t[1]),
            .zeroext_t(zeroext_t[1]), .ret_t(ret_t[1]),
            .branch(branch[1]),    .brdisp(brdisp[15:8]),    .selpc(selpc[1]),
.zeropc(zeropc[1]),
            .dmapc(dmapc[1]), .pc_ce(pc_ce[1]), .ret_ce(ret_ce[1]),
            .in_add(c_add),         .out_add(c_add[1]),         .in_ci(c_ci),
.out_ci(c_ci[1]), .in_branch(c_branch), .out_branch(c_branch[1]),
            .in_sum_t(c_sum_t),                          .out_sum_t(c_sum_t[1]),
.in_logic_t(c_logic_t), .out_logic_t(c_logic_t[1]),
```

```
            .in_shl_t(c_shl_t),   .out_shl_t(c_shl_t[1]),   .in_shr_t(c_shr_t),
.out_shr_t(c_shr_t[1]),
            .in_zeroext_t(c_zeroext_t),        .out_zeroext_t(c_zeroext_t[1]),
.in_ret_t(c_ret_t), .out_ret_t(c_ret_t[1]),
            .in_if_ir(c_if_ir), .out_if_ir(c_if_ir[31:16]),
            .in_ir(c_ir),       .out_ir(c_ir[31:16]),       .in_ex_ir(c_ex_ir),
.out_ex_ir(c_ex_ir[31:16]),
            .in_ex_call(c_ex_call),                .out_ex_call(c_ex_call[1]),
.in_ex_st(c_ex_st), .out_ex_st(c_ex_st[1]),
            .in_ifetch(c_ifetch),   .out_ifetch(c_ifetch[1]),   .in_dma(c_dma),
.out_dma(c_dma[1]), .in_sync_reset(c_sync_reset),
            .out_sync_reset(c_sync_reset[1]),         .in_dc_annul(c_dc_annul),
.out_dc_annul(c_dc_annul[1]),
            .in_ex_annul(c_ex_annul),            .out_ex_annul(c_ex_annul[1]),
.in_int_pend(c_int_pend), .out_int_pend(c_int_pend[1]),
            .in_dc_int(c_dc_int),                       .out_dc_int(c_dc_int[1]),
.in_dma_pend(c_dma_pend), .out_dma_pend(c_dma_pend[1]),
            .in_zero_pend(c_zero_pend), .out_zero_pend(c_zero_pend[1]),
            .inum(2'd2),.errorbus(errorbus));

            control ctrl3(
            .clk(clk), .rst(rst), .rdy(rdy),
            .int_req(int_req[2]), .dma_req(dma_req[2]), .zerodma(zerodma[2]),
            .insn(insn[47:32]), .a15(a15[2]), .z(z[2]), .n(n[2]), .co(co[2]),
.v(v[2]),
            .mem_ce(mem_ce[2]),                          .word_nxt(word_nxt[2]),
.read_nxt(read_nxt[2]),
            .dbus_nxt(dbus_nxt[2]), .dma(dma[2]),
            .rf_we(rf_we[2]),         .rna(rna[11:8]),         .rnb(rnb[11:8]),
.rdest(rdest[11:8]),
            .fwd(fwd[2]),       .imm(imm[35:24]),       .sextimm4(sextimm4[2]),
.zextimm4(zextimm4[2]),
            .wordimm4(wordimm4[2]), .imm12(imm12[2]),
            .pipe_ce(pipe_ce[2]), .b15_4_ce(b15_4_ce[2]),
            .add(add[2]), .ci(ci[2]), .logicop(logicop[5:4]), .sri(sri[2]),
            .sum_t(sum_t[2]),      .logic_t(logic_t[2]),      .shl_t(shl_t[2]),
.shr_t(shr_t[2]),
            .zeroext_t(zeroext_t[2]), .ret_t(ret_t[2]),
            .branch(branch[2]),    .brdisp(brdisp[23:16]),    .selpc(selpc[2]),
.zeropc(zeropc[2]),
            .dmapc(dmapc[2]), .pc_ce(pc_ce[2]), .ret_ce(ret_ce[2]),
            .in_add(c_add),          .out_add(c_add[2]),          .in_ci(c_ci),
.out_ci(c_ci[2]), .in_branch(c_branch), .out_branch(c_branch[2]),
            .in_sum_t(c_sum_t),                        .out_sum_t(c_sum_t[2]),
.in_logic_t(c_logic_t), .out_logic_t(c_logic_t[2]),
            .in_shl_t(c_shl_t),   .out_shl_t(c_shl_t[2]),   .in_shr_t(c_shr_t),
.out_shr_t(c_shr_t[2]),
            .in_zeroext_t(c_zeroext_t),        .out_zeroext_t(c_zeroext_t[2]),
.in_ret_t(c_ret_t), .out_ret_t(c_ret_t[2]),
            .in_if_ir(c_if_ir), .out_if_ir(c_if_ir[47:32]),
            .in_ir(c_ir),       .out_ir(c_ir[47:32]),       .in_ex_ir(c_ex_ir),
.out_ex_ir(c_ex_ir[47:32]),
            .in_ex_call(c_ex_call),                .out_ex_call(c_ex_call[2]),
.in_ex_st(c_ex_st), .out_ex_st(c_ex_st[2]),
            .in_ifetch(c_ifetch),   .out_ifetch(c_ifetch[2]),   .in_dma(c_dma),
.out_dma(c_dma[2]), .in_sync_reset(c_sync_reset),
            .out_sync_reset(c_sync_reset[2]),         .in_dc_annul(c_dc_annul),
.out_dc_annul(c_dc_annul[2]),
            .in_ex_annul(c_ex_annul),            .out_ex_annul(c_ex_annul[2]),
.in_int_pend(c_int_pend), .out_int_pend(c_int_pend[2]),
            .in_dc_int(c_dc_int),                       .out_dc_int(c_dc_int[2]),
.in_dma_pend(c_dma_pend), .out_dma_pend(c_dma_pend[2]),
```

```
            .in_zero_pend(c_zero_pend), .out_zero_pend(c_zero_pend[2]),
            .inum(2'd3),.errorbus(errorbus));

//due to partitioning requirements the tristate data bus drivers were moved to
this level
//evidently wired-or nets are exempt

            assign d[7:0] = (drivelo[0]) ? my_res[7:0] : 8'bz;
            assign d[15:8] = (drivehi[0]) ? my_res[15:8] : 8'bz;
            assign d[23:16] = (drivelo[1]) ? my_res[23:16] : 8'bz;
            assign d[31:24] = (drivehi[1]) ? my_res[31:24] : 8'bz;
            assign d[39:32] = (drivelo[2]) ? my_res[39:32] : 8'bz;
            assign d[47:40] = (drivehi[2]) ? my_res[47:40] : 8'bz;

endmodule

// original code by Jan Gray below


/* xr16.v -- xr16 pipelined RISC processor synthesizable Verilog model
 *
 * Copyright (C) 1999, 2000, Gray Research LLC.  All rights reserved.
 * The contents of this file are subject to the XSOC License Agreement;
 * you may not use this file except in compliance with this Agreement.
 * See the LICENSE file.
 *
 * $Header: /dist/xsocv/xr16.v 7     4/06/00 10:55a Jan $
 * $Log: /dist/xsocv/xr16.v $
 *
 * 7     4/06/00 10:55a Jan
 * polish


module xr16(
      clk, rst, rdy, ud_t, ld_t, udlt_t, int_req, dma_req, zerodma,
    insn, mem_ce, word_nxt, read_nxt, dbus_nxt, dma, addr_nxt, d);

      parameter          W      = 16;  // word width
      parameter          N      = W-1; // msb index

      input              clk;          // global clock
      input              rst;          // global async reset
      input              rdy;          // current memory access is ready
      input              ud_t;         //  active  low  store  data  MSB  output
enable
      input              ld_t;         //  active  low  store  data  LSB  output
enable
      input              udlt_t;              // active low store data MSB->LSB
output en
      input              int_req;      // interrupt request
      input              dma_req;      // DMA request
      input              zerodma;      // zero DMA counter request
      input  [N:0]  insn;        // new instruction word
      output             mem_ce;              // memory access clock enable
      output             word_nxt;     // next access is word wide
      output             read_nxt;     // next access is read
      output             dbus_nxt;     // next access uses on-chip data bus
      output             dma;          // current access is a DMA transfer
      output [N:0]  addr_nxt;    // address of next memory access
      inout  [N:0]  d;                 // on-chip data bus

      // locals
```

```
wire          a15;          // A operand msb
wire          z;                   // zero result condition code
wire          n;                   // negative result condition code
wire          co;                  // carry-out result codition code
wire          v;                   // oVerflow result condition code
wire          rf_we;        // register file write enable
wire [3:0]    rna;          // register file port A register number
wire [3:0]    rnb;          // register file port B register number
wire          fwd;          // forward result bus into A operand register
wire [11:0]   imm;          // 12-bit immediate field
wire          sextimm4;     // sign-extend 4-bit immediate operand
wire          zextimm4;     // zero-extend 4-bit immediate operand
wire          wordimm4;     // word-offset 4-bit immediate operand
wire          imm12;        // 12-bit immediate operand
wire          pipe_ce;      // pipeline clock enable
wire          b15_4_ce;     // b[15:4] clock enable
wire          add;          // 1 => A + B; 0 => A - B
wire          ci;                  // carry-in
wire [1:0]    logicop;      // logic unit opcode
wire          sri;          // shift right msb input
wire          sum_t;        // active low adder output enable
wire          logic_t;      // active low logic unit output enable
wire          zeroext_t;    // active low zero-extension output enable
wire          shr_t;        // active low shift right output enable
wire          shl_t;        // active low shift left output enable
wire          ret_t;        // active low return address output enable
wire          branch;              // branch taken
wire [7:0]    brdisp;              // 8-bit branch displacement
wire          selpc;        // address mux selects next PC
wire          zeropc;              // force next PC to 0
wire          dmapc;        // use DMA register in PC register file
wire          pc_ce;        // PC clock enable
wire          ret_ce;              // return address clock enable

// submodules
control ctrl(
       .clk(clk), .rst(rst), .rdy(rdy),
       .int_req(int_req), .dma_req(dma_req), .zerodma(zerodma),
       .insn(insn), .a15(a15), .z(z), .n(n), .co(co), .v(v),
       .mem_ce(mem_ce), .word_nxt(word_nxt), .read_nxt(read_nxt),
       .dbus_nxt(dbus_nxt), .dma(dma),
       .rf_we(rf_we), .rna(rna), .rnb(rnb),
       .fwd(fwd), .imm(imm), .sextimm4(sextimm4), .zextimm4(zextimm4),
       .wordimm4(wordimm4), .imm12(imm12),
       .pipe_ce(pipe_ce), .b15_4_ce(b15_4_ce),
       .add(add), .ci(ci), .logicop(logicop), .sri(sri),
       .sum_t(sum_t), .logic_t(logic_t), .shl_t(shl_t), .shr_t(shr_t),
       .zeroext_t(zeroext_t), .ret_t(ret_t),
       .branch(branch), .brdisp(brdisp), .selpc(selpc), .zeropc(zeropc),
       .dmapc(dmapc), .pc_ce(pc_ce), .ret_ce(ret_ce));

datapath dp(
       .clk(clk), .rst(rst),
       .rf_we(rf_we), .rna(rna), .rnb(rnb),
       .fwd(fwd), .imm(imm), .sextimm4(sextimm4), .zextimm4(zextimm4),
       .wordimm4(wordimm4), .imm12(imm12),
       .pipe_ce(pipe_ce), .b15_4_ce(b15_4_ce),
       .add(add), .ci(ci), .logicop(logicop), .sri(sri),
       .sum_t(sum_t), .logic_t(logic_t), .shl_t(shl_t), .shr_t(shr_t),
       .zeroext_t(zeroext_t), .ret_t(ret_t),
       .ld_t(ld_t), .ud_t(ud_t), .udlt_t(udlt_t),
       .branch(branch), .brdisp(brdisp), .selpc(selpc), .zeropc(zeropc),
```

```
            .dmapc(dmapc), .pc_ce(pc_ce), .ret_ce(ret_ce),
            .a15(a15), .z(z), .n(n), .co(co), .v(v),
            .addr_nxt(addr_nxt), .res(d));

endmodule
*/
```

U. XSCOUNTER.V

```
`timescale 1ns / 1ps
//`define CTR64BIT  uncomment for 64 bit timer
//`define CTR48BIT  uncomment this line for 64 or 48 bit timer
`include "voterids.v"
`define WORD_0_ADDR 5'h00
`define WORD_1_ADDR 5'h02
`define WORD_2_ADDR 5'h04
`define WORD_3_ADDR 5'h06
`define WORD_4_ADDR 5'h08
`define WORD_5_ADDR 5'h0A
`define WORD_6_ADDR 5'h0C
`define WORD_7_ADDR 5'h0E
`define INT_ACK           5'h10
`define INT_ENABLE  5'h11
`define INT0_ENABLE 5'h12
`define INT1_ENABLE 5'h13
`define INT2_ENABLE 5'h14
`define INT3_ENABLE 5'h15
`define INT4_ENABLE 5'h16
`define INT5_ENABLE 5'h17
`define INT6_ENABLE 5'h18
`define INT7_ENABLE 5'h19
`define TIMER_MASK  5'h1A
`define TIMER_ACK         5'h1C

`define INT_DELAY_CLK     5
`define TIMER_INT         3


module xscounter64_i (clk, rst, addr, din, dout, errorbus,
                              out_count0, out_count1,
                              count0, count1,
`ifdef CTR64BIT
                              out_count3,count3,
`endif
`ifdef CTR48BIT
                              out_count2, count2,
`endif
                              ld_t, ld_ce, int_req_in, int_req_out);


      input               clk;
      input               rst;
      input [14:0] addr;
      input [47:0] din;
      output [15:0]       dout;
      inout [15:0] errorbus;
      input [2:0]         ld_t;        // driving data to the bus
                                                           // if it is the LSW
then latch the rest into the temp registers
      output [15:0]       out_count0;
```

```
	output [15:0]	out_count1;

	input [47:0] count0;
	input [47:0] count1;

`ifdef CTR64BIT
	output [15:0]	out_count3;
	input [47:0] count3;
`endif
`ifdef CTR48BIT
	output [15:0]	out_count2;
	input [47:0] count2;
`endif
	input [2:0]		ld_ce;
	input [7:0]	int_req_in;
	output			int_req_out;

	wor [15:0]		errorbus;

	reg [15:0]		word_1, word_5;


	`ifdef CTR64BIT
		reg [63:0]		xs_count;
	`elsif CTR48BIT
		reg [47:0]		xs_count;
	`else
		reg [31:0]		xs_count;
	`endif

   `ifdef CTR64BIT
		reg [15:0]		word_3;
		reg [15:0]		word_7;
	`endif
	`ifdef CTR48BIT
		reg [15:0]		word_2, word_6;
	`endif

	reg [15:0]		dout;

	wire [7:0]		int_req_in;
	reg				int_req_out;

	reg				int_idle=1'b1;
	reg				int_now=1'b0;
	reg				ack_wait=1'b0;
	reg				int_delay=1'b0;
	reg				delay_pend=1'b0;

	reg [3:0]		delay_cnt=4'b0;

	reg				int_enable=1'b0;  // global enable/disable for
interrupts
	reg				ien_0=1'b0;
	reg				ien_1=1'b0;
	reg				ien_2=1'b0;
	reg				ien_3=1'b0;
	reg				ien_4=1'b0;
	reg				ien_5=1'b0;
	reg				ien_6=1'b0;
	reg				ien_7=1'b0;
```

```
      reg [15:0]          timer_mask;
      reg                       timer_state0;
      reg                       timer_state1;
      reg                       timer_state2;

      wire                      timer_interrupt;
      wire                      timer_now;

      wire [63:0]  v_xs_count;
      wire                      got_interrupt;
      wire [4:0]          v_addr;
      wire [15:0]         v_din;
      wire                      v_ld_t;
      wire                      v_ld_ce;

      assign out_count0 = xs_count[15:0];
      assign out_count1 = xs_count[31:16];
      `ifdef CTR48BIT
      assign out_count2 = xs_count[47:32];
      `endif

      `ifdef CTR64BIT
      assign out_count3 = xs_count[63:48];
      `endif

      assign got_interrupt = int_enable & ((ien_0 & int_req_in[0]) | (ien_1 &
int_req_in[1])
                                                                 |      (ien_2      &
int_req_in[2]) | (ien_3 & timer_interrupt)
                                                                 |      (ien_4      &
int_req_in[4]) | (ien_5 & int_req_in[5])
                                                                 |      (ien_6      &
int_req_in[6]) | (ien_7 & int_req_in[7]));

      voter1nsyn #(1)     ld_ce_v         (.din(ld_ce),         .dout(v_ld_ce),
.errorbus(errorbus),
                                .clk(clk), .rst(rst), .ID(`COUNT64_LD_CE));
      voter1nsyn #(1)     ld_t_v          (.din(ld_t),          .dout(v_ld_t),
.errorbus(errorbus),
                                .clk(clk), .rst(rst), .ID(`COUNT64_LD_T));
      voter1nsyn #(5)     addr_v          (.din(addr),          .dout(v_addr),
.errorbus(errorbus),
                                .clk(clk), .rst(rst), .ID(`COUNT64_ADDR));
      voter1nsyn #(16)    din_v (.din(din), .dout(v_din),  .errorbus(errorbus),
                                .clk(clk), .rst(rst), .ID(`COUNT64_DIN));
      voter1nsyn #(16)    count0_v (.din(count0), .dout(v_xs_count[15:0]),
                                .errorbus(errorbus),   .clk(clk),   .rst(rst),
.ID(`COUNT64_COUNT0));
      voter1nsyn #(16)    count1_v (.din(count1), .dout(v_xs_count[31:16]),
                                .errorbus(errorbus),   .clk(clk),   .rst(rst),
.ID(`COUNT64_COUNT1));
      `ifdef CTR48BIT
      voter1nsyn #(16)    count2_v (.din(count2), .dout(v_xs_count[47:32]),
                                .errorbus(errorbus),   .clk(clk),   .rst(rst),
.ID(`COUNT64_COUNT2));
      `endif
      `ifdef CTR64BIT
      voter1nsyn #(16)    count3_v (.din(count3), .dout(v_xs_count[63:48]),
                                .errorbus(errorbus),   .clk(clk),   .rst(rst),
.ID(`COUNT64_COUNT3));
      `endif
```

```
	// begin timer section
	assign timer_interrupt = timer_state1;
	assign timer_now = |(timer_mask & v_xs_count[31:16]);

	always @(posedge clk or posedge rst) begin
		if (rst) begin
			timer_state0 <= 1;   // idle
			timer_state1 <= 0;   //interrupt active
			timer_state2 <= 0;    //interrupt acked but timer bit still
high, wait for low
		end
		else begin
			if (timer_state0 & timer_now) begin  // go to state 1
				timer_state0 <= 0;
				timer_state1 <= 1;
			end
			if (timer_state1 & v_addr == `TIMER_ACK && v_ld_ce) begin //
got ack
				if (timer_now) begin
					timer_state1 <= 0;  // got ack timer bit still
high, wait for low
					timer_state2 <= 1;
				end
				else begin                  // got ack and timer is
already low, go to idle
					timer_state1 <= 0;
					timer_state0 <= 1;
				end
			end
			if (timer_state2 & ~timer_now) begin   // waiting for low
timer active, got it
				timer_state2 <= 0;                              // now
idle state 0
				timer_state0 <= 1;
			end
		end
	end

	always @(posedge clk or posedge rst) begin
		if (rst) begin
			timer_mask <= 0;
		end
		else begin
			if (v_ld_ce && v_addr == `TIMER_MASK) begin
				timer_mask <= v_din;
			end
		end
	end

	// end timer section


	// begin interrupt section
	always @(posedge clk or posedge rst)
		begin
		if (rst) begin
			int_idle <= 1;
			int_now <= 0;
			ack_wait <= 0;
			int_delay <= 0;
			delay_cnt <= 0;
```

```
				int_req_out <= 0;
			end
			else begin
				if(int_idle) begin  /* state 0, no interrupts in process */
					if (got_interrupt) begin
						int_idle <= 0;
						int_now <= 1;
						int_req_out <= 1;
					end
				end
				if(int_now)  begin   /*  got  an  interrupt  request,  signal 
processor */
					int_req_out <= 0;
					int_now <= 0;
					ack_wait <= 1;
				end
				if(ack_wait) begin   /* now waiting on an acknowledge from 
the processor for the interrupt */
					if(ld_ce && (v_addr == `INT_ACK)) begin
						ack_wait <= 0;
						int_delay <= 1;
						delay_cnt <= 0;
					end
				end
				if(int_delay)  begin   /*  delay  INT  DELAY  CLOCKS  before 
processing interrupts again */
					if (delay_cnt < `INT_DELAY_CLK) begin
						delay_cnt <= delay_cnt + 1;

					end
					else begin
						int_delay <= 0;
						int_idle <= 1;
					end
				end
			end
		end


		// writes for interrupt acknowledge and enables
		always @(posedge clk or posedge rst) begin
			if(rst) begin
				int_enable <= 0;
				ien_0 <= 0;
				ien_1 <= 0;
				ien_2 <= 0;
				ien_3 <= 0;
				ien_4 <= 0;
				ien_5 <= 0;
				ien_6 <= 0;
				ien_7 <= 0;
			end
			else begin
			if (v_ld_ce) begin
				if (v_addr == `INT_ENABLE)
					int_enable <= v_din[0];
				if (v_addr == `INT0_ENABLE)
					ien_0 <= v_din[0];
				if (v_addr == `INT1_ENABLE)
					ien_1 <= v_din[0];
				if (v_addr == `INT2_ENABLE)
					ien_2 <= v_din[0];
```

```
                        if (v_addr ==       `INT3_ENABLE)
                                ien_3 <= v_din[0];
                        if (v_addr ==       `INT4_ENABLE)
                                ien_4 <= v_din[0];
                        if (v_addr ==       `INT5_ENABLE)
                                ien_5 <= v_din[0];
                        if (v_addr ==       `INT6_ENABLE)
                                ien_6 <= v_din[0];
                        if (v_addr ==       `INT7_ENABLE)
                                ien_7 <= v_din[0];
                        end
                end
        end

        // end interrupt section

        // begin counter section

        always @(*) begin
                if (rst)
                        dout <= 16'b0;
                else begin
                case (v_addr)
                        `WORD_0_ADDR:
                                dout <= xs_count[15:0];
                        `WORD_1_ADDR:
                                dout <= word_1;
                        `WORD_2_ADDR:
`ifdef CTR48BIT
                                dout <= word_2;
`else
                                dout <= 0;
`endif
                        `WORD_3_ADDR:
`ifdef CTR64BIT
                                dout <= word_3;
`else // 48BIT
                                dout <= 0;
`endif
                        `WORD_4_ADDR:
                                dout <= xs_count[15:0];
                        `WORD_5_ADDR:
                                dout <= word_5;
                        `WORD_6_ADDR:
`ifdef CTR48BIT
                                dout <= word_6;
`else
                                dout <= 0;
`endif
                        `WORD_7_ADDR:
`ifdef CTR64BIT
                                dout <= word_7;
`else
                                dout <= 0;
`endif
                        default:
                                dout <= 16'b0;
                endcase
                end
        end

        always @(posedge clk or posedge rst) begin
```

```
            if(rst)
                  begin
                        xs_count <= 0;
                        word_1 <= 0;
                        word_5 <= 0;
`ifdef CTR48BIT
                        word_2 <= 0;
                        word_6 <= 0;
`endif
`ifdef CTR64BIT
                        word_3 <= 0;
                        word_7 <= 0;
`endif
                  end
            else
                  begin
                        xs_count <= v_xs_count + 1;
                        if (~v_ld_t && (addr==`WORD_0_ADDR)) begin
                              word_1 <= v_xs_count[31:16];
`ifdef CTR48BIT
                              word_2 <= v_xs_count[47:32];
`endif
`ifdef CTR64BIT
                              word_3 <= v_xs_count[63:48];
`endif
                        end
                        if (~v_ld_t && (addr==`WORD_4_ADDR)) begin
                              word_5 <= v_xs_count[31:16];
`ifdef CTR48BIT
                              word_6 <= v_xs_count[47:32];
`endif
`ifdef CTR64BIT
                              word_7 <= v_xs_count[63:48];
`endif
                        end
                  end
      end

      //end counter section

endmodule



module xscounter64(clk, rst, ctrl, int_req_in, int_req_out, sel, d, errorbus);
    input clk;
    input rst;
    input [47:0] ctrl;
    input [2:0] sel;
        input [23:0] int_req_in;
        output [2:0] int_req_out;
    inout [47:0] d;
        inout [15:0] errorbus;

      tri    [47:0] d;
      wor    [15:0] errorbus;

      wire [14:0]  addr;
      wire [2:0]         ud_t, ld_t, ud_ce, ld_ce;
      wire [47:0] dout;
      wire [47:0] count0, count1; // 64 bits would add too much delay
`ifdef CTR48BIT
```

```
      wire [47:0] count2;
`endif
`ifdef CTR64BIT
      wire [47:0] count3; // divide count voter into 4 parts
`endif

      ctrl_dec_syn   dec0(.clk(clk),   .rst(rst),   .ctrl(ctrl),   .sel(sel),
.ud_t(ud_t[0]),
                         .ld_t(ld_t[0]), .ud_ce(ud_ce[0]), .ld_ce(ld_ce[0]),
.addr(addr[4:0]),
                         .errorbus(errorbus), .id(`COUNT64_CTRL_DEC));

      ctrl_dec_syn   dec1(.clk(clk),   .rst(rst),   .ctrl(ctrl),   .sel(sel),
.ud_t(ud_t[1]),
                         .ld_t(ld_t[1]), .ud_ce(ud_ce[1]), .ld_ce(ld_ce[1]),
.addr(addr[9:5]),
                         .errorbus(errorbus), .id(`COUNT64_CTRL_DEC));


      ctrl_dec_syn   dec2(.clk(clk),   .rst(rst),   .ctrl(ctrl),   .sel(sel),
.ud_t(ud_t[2]),
                         .ld_t(ld_t[2]), .ud_ce(ud_ce[2]), .ld_ce(ld_ce[2]),
.addr(addr[14:10]),
                         .errorbus(errorbus), .id(`COUNT64_CTRL_DEC));

      xscounter64_i xscounter64_i0(.clk(clk), .rst(rst), .addr(addr), .din(d),
.dout(dout[15:0]),
                               .errorbus(errorbus),.ld_t(ld_t),
.out_count0(count0[15:0]),
                               .out_count1(count1[15:0]),
                               .count0(count0),.count1(count1),
`ifdef CTR48BIT
                               .count2(count2),.out_count2(count2[15:0]),
`endif
`ifdef CTR64BIT
                               .count3(count3),.out_count3(count3[15:0]),
`endif
                               .ld_ce(ld_ce[0]),
.int_req_in(int_req_in[7:0]), .int_req_out(int_req_out[0]));

      xscounter64_i xscounter64_i1(.clk(clk), .rst(rst), .addr(addr), .din(d),
.dout(dout[31:16]),
                               .errorbus(errorbus),.ld_t(ld_t),
.out_count0(count0[31:16]),
                               .out_count1(count1[31:16]),
                               .count0(count0), .count1(count1),
`ifdef CTR48BIT
                               .count2(count2), .out_count2(count2[31:16]),
`endif
`ifdef CTR64BIT
                               .count3(count3),.out_count3(count3[31:16]),
`endif
                               .ld_ce(ld_ce[1]),
.int_req_in(int_req_in[15:8]), .int_req_out(int_req_out[1]));

      xscounter64_i xscounter64_i2(.clk(clk), .rst(rst), .addr(addr), .din(d),
.dout(dout[47:32]),
                               .errorbus(errorbus),.ld_t(ld_t),
.out_count0(count0[47:32]),
                               .out_count1(count1[47:32]),
                               .count0(count0), .count1(count1),
`ifdef CTR48BIT
```

```
                                .count2(count2),.out_count2(count2[47:32]),
`endif
`ifdef CTR64BIT
                                .out_count3(count3[47:32]), .count3(count3),
`endif
                                .ld_ce(ld_ce[2]),
.int_req_in(int_req_in[23:16]), .int_req_out(int_req_out[2]));

	assign d[7:0] = ld_t[0] ? 8'bz : dout[7:0];
	assign d[15:8] = ud_t[0] ? 8'bz : dout[15:8];
	assign d[23:16] = ld_t[1] ? 8'bz : dout[23:16];
	assign d[31:24] = ud_t[1] ? 8'bz : dout[31:24];
	assign d[39:32] = ld_t[2] ? 8'bz : dout[39:32];
	assign d[47:40] = ud_t[2] ? 8'bz : dout[47:40];

endmodule
```

V. XSOC.V

```
/* xsoc.v -- XSOC System-on-a-Chip synthesizable Verilog model
 *
 * Copyright (C) 1999, 2000, Gray Research LLC.  All rights reserved.
 * The contents of this file are subject to the XSOC License Agreement;
 * you may not use this file except in compliance with this Agreement.
 * See the LICENSE file.
 *
 * $Header: /dist/xsocv/xsoc.v 8     4/06/00 10:55a Jan $
 * $Log: /dist/xsocv/xsoc.v $
 *
 * 8     4/06/00 10:55a Jan
 * polish
 */
 /* last modified by David Dwiggins, 24 August 2008.  Adapted to use
 *  TMR/ECC functionality for fault-tolerance.  */
 /* original version by Jan Gray included at the end of file */
`timescale 1ns / 1ps

module xsoc(clk, rst, ctrl, ctrl0, sel, d, int_req, errorbus, addr_nxt);

	parameter		W	= 16;  // word width
	parameter		N	= W-1; // msb index
	parameter		XAN	= 16;  // external address msb index

	// ports
	input			clk;		// global clock
	input			rst;		// global async reset
	output [47:0]	ctrl;
	input   [2:0]	ctrl0;		// insert wait states for IO
	output [23:0]	sel;
	inout   [47:0]	d;
	input  [2:0]	int_req;
	inout  [15:0]	errorbus;
	output [47:0]	addr_nxt;

	// locals
	wire [47:0]	addr_nxt;	// address of next memory access
	wire [47:0]	d;			// on-chip data bus
	wire [47:0]	xd;		// RAM bus
	wire   [2:0]	mem_ce;			// memory access clock enable
```

```
	wire   [2:0]  word_nxt;     // next access is word wide
	wire   [2:0]  read_nxt;     // next access is read
	wire   [2:0]  dbus_nxt;     // next access uses on-chip data bus
	wire   [2:0]         sub;          // ram store lsb into upper byte
	wire   [2:0]         slb;          // ram store lower byte
	wire   [2:0]         rub;          // ram read upper byte into data bus lsb
	wire   [2:0]         we;           // ram write enable upper
	wire   [2:0]         uxd_t; // RAM MSB data out enable
	wire   [2:0]         lxd_t;    // RAM LSB data out enable

	wire   [2:0]  dma;          // current access is a DMA transfer
	wire   [2:0]         dma_req;      // DMA request
	wire   [2:0]         zerodma;      // zero DMA counter request
	wire   [2:0]         rdy;          // current memory access is ready
	wire   [2:0]         ud_t;         //  active  low  store  data  MSB  output
enable
	wire   [2:0]         ld_t;         //  active  low  store  data  LSB  output
enable
	wire   [2:0]         udlt_t;             // active low store data MSB->LSB
output enable
	wire [23:0]          sel;          // on-chip peripheral select
	wire [47:0]          ctrl;         // abstract control bus
	wire   [47:0]        xa;           // RAM address register
	wire [47:0]  areg, breg;
	wire   [2:0]  pipe_ce;
	wire   [47:0]        memdta;
	wire   [47:0] mem_addr; // memory next address
	wire   [47:0]        if_ir, ir, ex_ir;
	wire   [2:0]   fwd;
	wire   [35:0] imm;
	wire [2:0] sextimm4;
	wire [2:0] zextimm4;
	wire [2:0] wordimm4;
	wire [2:0] imm12;
	wire [2:0] drivelo;
	wire [2:0] drivehi;
	wor    [15:0] errorbus;
	wire   [17:0] syndrome;
	wire  [2:0]    ctrl0;
	wire   [2:0]         int_req;      // interrupt request
	wire   [2:0]         vack;
	wire   [2:0]         vreset;
	wire   [2:0]         vreq;

	assign vreset = 3'b0;  // no vga so disable DMA
	assign vreq = 3'b0;

	// submodules
	xr16 p(
		.clk(clk), .rst(rst), .rdy(rdy),
		.ud_t(ud_t), .ld_t(ld_t), .udlt_t(udlt_t),
		.int_req(int_req), .dma_req(dma_req), .zerodma(zerodma),
		.insn(xd), .mem_ce(mem_ce),
		.word_nxt(word_nxt), .read_nxt(read_nxt), .dbus_nxt(dbus_nxt),
		.dma(dma),  .addr_nxt(addr_nxt),  .d(d),  .areg(areg),  .breg(breg),
.pipe_ce(pipe_ce),
		.memdta(memdta),.errorbus(errorbus),
		.if_ir(if_ir), .ir(ir), .ex_ir(ex_ir),
		.fwd(fwd),  .imm(imm),  .sextimm4(sextimm4),  .zextimm4(zextimm4),
.wordimm4(wordimm4), .imm12(imm12),
		.drivelo(drivelo), .drivehi(drivehi));
```

```
	// ECC Block RAM

	bram_ecc bram_ecc1(.clk(clk), .rst(rst), .sub(sub), .slb(slb), .rub(rub),
		.uxd_t(uxd_t),		.lxd_t(lxd_t),		.memdta(memdta),
.mem_addr(mem_addr), .we(we),
		.d(d),	.xd(xd),	.errorbus(errorbus),	.mem_ce(mem_ce),
.addr_nxt(addr_nxt));

	memctrl mc1(
			.clk(clk), .rst(rst),
			.mem_ce(mem_ce), .word_nxt(word_nxt), .read_nxt(read_nxt),
			.dbus_nxt(dbus_nxt), .dma(dma), .addr_nxt(addr_nxt),
			.dma_req_in(vreq), .zero_req_in(vreset),.ctrl0(ctrl0[0]),
			.rdy(rdy[0]),		.ud_t(ud_t[0]),		.ld_t(ld_t[0]),
.udlt_t(udlt_t[0]),
			.dma_req(dma_req[0]), .zerodma(zerodma[0]),
			.uxd_t(uxd_t[0]),	.lxd_t(lxd_t[0]),	.store_ub(sub[0]),
.store_lb(slb[0]), .read_ub(rub[0]),
			.write_en(we[0]),
			.sel(sel[7:0]), .ctrl(ctrl[15:0]), .dma_ack(vack[0]),
			.errorbus(errorbus));

	memctrl mc2(
			.clk(clk), .rst(rst),
			.mem_ce(mem_ce), .word_nxt(word_nxt), .read_nxt(read_nxt),
			.dbus_nxt(dbus_nxt), .dma(dma), .addr_nxt(addr_nxt),
			.dma_req_in(vreq), .zero_req_in(vreset),.ctrl0(ctrl0[1]),
			.rdy(rdy[1]),		.ud_t(ud_t[1]),		.ld_t(ld_t[1]),
.udlt_t(udlt_t[1]),
			.dma_req(dma_req[1]), .zerodma(zerodma[1]),
			.uxd_t(uxd_t[1]),	.lxd_t(lxd_t[1]),	.store_ub(sub[1]),
.store_lb(slb[1]), .read_ub(rub[1]),
			.write_en(we[1]),
			.sel(sel[15:8]), .ctrl(ctrl[31:16]), .dma_ack(vack[1]),
			.errorbus(errorbus));

	memctrl mc3(
			.clk(clk), .rst(rst),
			.mem_ce(mem_ce), .word_nxt(word_nxt), .read_nxt(read_nxt),
			.dbus_nxt(dbus_nxt), .dma(dma), .addr_nxt(addr_nxt),
			.dma_req_in(vreq), .zero_req_in(vreset),.ctrl0(ctrl0[2]),
			.rdy(rdy[2]),		.ud_t(ud_t[2]),		.ld_t(ld_t[2]),
.udlt_t(udlt_t[2]),
			.dma_req(dma_req[2]), .zerodma(zerodma[2]),
			.uxd_t(uxd_t[2]),	.lxd_t(lxd_t[2]),	.store_ub(sub[2]),
.store_lb(slb[2]), .read_ub(rub[2]),
			.write_en(we[2]),
			.sel(sel[23:16]), .ctrl(ctrl[47:32]), .dma_ack(vack[2]),
			.errorbus(errorbus));

	// data bus isn't driven all the time; floating inputs cause false
syndromes to be reported
	datapulldown datapulldown1(d);

	// make sure the errorbus is assigned
	assign errorbus=16'b0;

endmodule

// original code below

/* xsoc.v -- XSOC System-on-a-Chip synthesizable Verilog model
```

```
 *
 * Copyright (C) 1999, 2000, Gray Research LLC.  All rights reserved.
 * The contents of this file are subject to the XSOC License Agreement;
 * you may not use this file except in compliance with this Agreement.
 * See the LICENSE file.
 *
 * $Header: /dist/xsocv/xsoc.v 8     4/06/00 10:55a Jan $
 * $Log: /dist/xsocv/xsoc.v $
 *
 * 8     4/06/00 10:55a Jan
 * polish


module xsoc(
	clk, rst, par_d, par_s, r, g, b, hsync_n, vsync_n,
	ram_ce_n, ram_oe_n, ram_we_n, res8031, xa, xa_0, xd);

	parameter		W	= 16;	// word width
	parameter		N	= W-1;	// msb index
	parameter		XAN	= 16;	// external address msb index

	// ports
	input			clk;		// global clock
	input			rst;		// global async reset
	input  [4:0]	par_d;		// parallel port data in
	output [6:3]	par_s;		// parallel port status out
	output [1:0]	r, g, b;	// (red, green, blue) outputs
	output			hsync_n;	// active low horizontal sync output
	output			vsync_n;	// active low vertical sync output
	output			ram_ce_n;	// active low external RAM chip enable
	output			ram_oe_n;	// active low external RAM output enable
	output			ram_we_n;	// active low external RAM write enable
	output			res8031;	// reset external 8031 MCU
	output [XAN:0]	xa;			// external RAM address bus
	reg		[XAN:0]	xa;
	output			xa_0;		// external RAM address lsb
	inout  [7:0]	xd;			// external RAM data bus
	tri		[7:0]	xd;

	// locals
	wire [N:0]	addr_nxt;	// address of next memory access
	wire [N:0]	d;			// on-chip data bus
	wire		mem_ce;			// memory access clock enable
	wire		word_nxt;	// next access is word wide
	wire		read_nxt;	// next access is read
	wire		dbus_nxt;	// next access uses on-chip data bus
	wire		dma;		// current access is a DMA transfer
	wire		int_req;	// interrupt request
	wire		dma_req;	// DMA request
	wire		zerodma;	// zero DMA counter request
	wire		rdy;		// current memory access is ready
	wire		ud_t;		// active low store data MSB output enable
	wire		ld_t;		// active low store data LSB output enable
	wire		udlt_t;			// active low store data MSB->LSB output
enable
	wire		vreq;		// video word DMA request
	wire		vreset;			// video word address counter reset
	wire		vack;		// video DMA acknowledge
	wire [7:0]	sel;		// on-chip peripheral select
	wire [15:0]	ctrl;		// abstract control bus
	reg  [7:0]	xdq;		//  RAM  external  data  half-cycle  capture
register
```

```
	// submodules
	xr16 p(
		.clk(clk), .rst(rst), .rdy(rdy),
		.ud_t(ud_t), .ld_t(ld_t), .udlt_t(udlt_t),
		.int_req(int_req), .dma_req(dma_req), .zerodma(zerodma),
		.insn({xdq,xd}), .mem_ce(mem_ce),
		.word_nxt(word_nxt), .read_nxt(read_nxt), .dbus_nxt(dbus_nxt),
		.dma(dma), .addr_nxt(addr_nxt), .d(d));

	memctrl mc(
		.clk(clk), .rst(rst),
		.mem_ce(mem_ce), .word_nxt(word_nxt), .read_nxt(read_nxt),
		.dbus_nxt(dbus_nxt), .dma(dma), .addr_nxt(addr_nxt),
		.dma_req_in(vreq), .zero_req_in(vreset),
		.rdy(rdy), .ud_t(ud_t), .ld_t(ld_t), .udlt_t(udlt_t),
		.int_req(int_req), .dma_req(dma_req), .zerodma(zerodma),
		.ram_oe_n(ram_oe_n), .ram_we_n(ram_we_n), .xa_0(xa_0),
		.xdout_t(xdout_t), .uxd_t(uxd_t), .lxd_t(lxd_t),
		.sel(sel), .ctrl(ctrl), .dma_ack(vack));

	vga vga(
		.clk(clk), .rst(rst), .vack(vack), .pixels_in({xdq,xd}),
	    .vreq(vreq), .vreset(vreset),
		.hsync_n(hsync_n), .vsync_n(vsync_n), .r(r), .g(g), .b(b));

	// External RAM interface
	always @(posedge clk or posedge rst) begin
		if (rst)
			xa <= 0;
		else if (mem_ce)
			xa[16:0] <= {1'b0, addr_nxt[15:0]};	// use low 64 KB of
SRAM
	end
	always @(negedge clk or posedge rst) begin
		if (rst)
			xdq <= 0;
		else
			xdq <= xd;
	end
	assign xd		= xdout_t ? 8'bz : d[7:0];
	assign d[7:0] = lxd_t ? 8'bz : xd;
	assign d[15:8]	= uxd_t ? 8'bz : xdq;
	assign ram_ce_n	= 0;
	assign res8031	= 1;

	// On-chip peripherals
	xram16x16 iram(
		.clk(clk), .rst(rst), .ctrl(ctrl), .sel(sel[0]), .d(d));

	xin8 parin(
		.clk(clk), .rst(rst), .ctrl(ctrl), .sel(sel[1]), .d(d[7:0]),
		.i({3'b0,par_d}));

	xout4 parout(
		.clk(clk), .rst(rst), .ctrl(ctrl), .sel(sel[2]), .d(d[3:0]),
		.q(par_s));

endmodule
 */
```

W. XSOC_PC104.V

```
`timescale 1ns / 1ps
`include "voterids.v"
// Code translated from Mindy Surratt's pc104IntArm.vhd
// and modified to work with the CFTP XSOC by David Dwiggins

//-- Constants, formerly in pc104IntPack.vhd
`define C_LowAddress                          10'b1101000000            //
0x340
`define C_HighAddress                         10'b1101101000            //
0x368
`define C_DATA_ADDR                           10'b1101000000            //
0x340
`define C_TEST_ADDR                           10'b1101000001            //
0x341
//`define C_RESET                         10'b1101000010          // 0x342
`define C_STATUS_ADDR                         10'b1101000010            //
0x342
`define C_INTACK_ADDR                         10'b1101000100            //
0x344
`define C_INTOFF_ADDR                         10'b1101000101            //
0x345
`define C_INTON_ADDR                          10'b1101000110            //
0x346

`define STATUS_REG_ALLOW_PROC_READ 0
`define STATUS_REG_BLOCK_PROC_WRITE 1
`define STATUS_REG_FIFO_ALMOST_FULL 2

`define      XS_PC104_DATA_ADDR        5'b00000                         //
0x00
`define      XS_PC104_FULL_ADDR        5'b00010                         //
0x02
`define      XS_PC104_EMPTY_ADDR       5'b00011                         //
       0x03
`define      XS_PC104_ALMOST_FULL_ADDR  5'b00100                 // 0x04
`define      XS_PC104_GOT_DATA                5'b00101
       // 0x05

module xsoc_pc104_mod(clk, rst, int_req, d, T_DATA_io, DATA_io_out,
             DATA_io_en,  T_ADDRESS_i,  T_IOREAD_i,  T_IOWRITE_i,  T_IOCS_i,
T_INTRPT_o,
             fifo_full,  dout,  fifo_wren,  fifo_rd,  fifo_rd_clk,  fifo_din,
fifo_dout,
             fifo_empty, fifo_almost_full, ld_t, ld_ce, addr, errorbus);
    input                 clk;
    input                 rst;
    output                int_req;
    input [47:0]    d;
    input [7:0]     T_DATA_io;
       output [7:0] DATA_io_out;
       output                    DATA_io_en;
    input [9:0]     T_ADDRESS_i;
    input                 T_IOREAD_i;
    input                 T_IOWRITE_i;
    input                 T_IOCS_i;
    output                T_INTRPT_o;
       input              fifo_full;
       output                    [15:0] dout;
       output [7:0] fifo_din;
```

```
    output                      fifo_wren;
    output                      fifo_rd;
    output                      fifo_rd_clk;
    input [7:0]  fifo_dout;
    input               fifo_empty;
    input               fifo_almost_full;
    input [2:0]  ld_t;
    input [2:0]  ld_ce;
    input [14:0] addr;
    inout [15:0] errorbus;

    wor   [15:0] errorbus;
    wire                clk;
wire                    rst;
wire [47:0]      d;
    wire                        int_req;
wire [7:0]       T_DATA_io;
wire [9:0]       T_ADDRESS_i;
wire                    T_IOREAD_i;
wire                    T_IOWRITE_i;
wire                    T_IOCS_i;

    reg T_INTRPT_o;

   // Signals that were external to the verilog PC/104 interface version
         // but are now attached to the XSOC interface code
         wire DATA_ACK_i;
         wire [7:0] DATA_i;
         reg [7:0] DATA_o;
         wire DATA_WREN_i;
         wire FIFO_FULL_o;

         reg S_RangeCheck;
         wire S_Decode;
         reg [7:0] S_IOTemp;

         // Data for IORead (top level writing to PC/104)
         wire [8:0] DATA_i_reg;
         wire [7:0]   status_reg_iord;

         // Flags to/from top level for receiving data from PC/104
         reg data_ack;
         reg data_rdy;

         // fifo signals
         wire                fifo_rd_clk;
         reg                 fifo_rd;
         wire                fifo_full; // not used
         wire                fifo_almost_full;
         wire                fifo_empty;
         wire                fifo_ainit; // not used
         wire [7:0] fifo_din;
         wire [7:0] fifo_dout;
         wire                fifo_wren;

         //INTRPT signals
         reg                 intrpt_ack;
         reg                 intrpt_on;
         reg                 intrpt_off;
         reg                 intrpt_active;
         reg [31:0]   intrpt_cnt;
```

```
            // Signals to clock PC/104 pins
            reg                 got_data;
            reg                 got_data_reg;
            reg                 wr_reset;
            reg [7:0]     s_data_i;
            reg [9:0]     s_address_i;

            reg                 iow_d;

            // XSOC Interface signals
            wire [2:0]    ld_t, ld_ce;
            wire [4:0]    v_addr;
            wire                v_ld_t, v_ld_ce;
            wire [7:0] v_d;

            reg                 [15:0] dout;

            assign int_req=data_rdy;

            voter1nsynh #(8)    d_v         (.din({d[39:32],d[23:16],d[7:0]}),
.dout(v_d), .errorbus(errorbus), .clk(clk), .rst(rst), .ID(`PC104_MO_D_V));
            voter1nsynh #(1)    ld_ce_v     (.din(ld_ce),      .dout(v_ld_ce),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID(`PC104_MO_LD_CE_V));
            voter1nsynh #(1)    ld_t_v      (.din(ld_t),       .dout(v_ld_t),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID(`PC104_MO_LD_T_V));
            voter1nsynh #(5)    addr_v      (.din(addr),       .dout(v_addr),
.errorbus(errorbus), .clk(clk), .rst(rst), .ID(`PC104_MO_ADDR_V));


            always  @(v_addr  or  DATA_o  or  fifo_full  or  fifo_empty  or
fifo_almost_full or data_rdy) begin
                  case (v_addr)
                        `XS_PC104_DATA_ADDR:
                              dout <= {8'b0,DATA_o[7:0]};
                        `XS_PC104_FULL_ADDR:
                              dout <= {15'b0,fifo_full};
                        `XS_PC104_EMPTY_ADDR:
                              dout <= {15'b0,fifo_empty};
                        `XS_PC104_ALMOST_FULL_ADDR:
                              dout <= {15'b0,fifo_almost_full};
                        `XS_PC104_GOT_DATA:
                              dout <= {15'b0,data_rdy};
                        default:
                              dout <= 16'b0;
                  endcase
            end


            // writes to ARM
            assign DATA_WREN_i = v_ld_ce;
            assign DATA_i = v_d[7:0];

            // data driven to the data bus, acknowledge receipt to ARM
            assign DATA_ACK_i = ~v_ld_t;  // active low signal

//
// Address Range Check lines
//

            always @(T_ADDRESS_i) begin
                  if   ((T_ADDRESS_i   >=   `C_LowAddress)   &&   (T_ADDRESS_i
<=`C_HighAddress))
```

```
                        S_RangeCheck <= 1;
                else
                        S_RangeCheck <= 0;
        end

//
// Decode Line
//
// New signal T_CARD_BLE0_i added 7 Jun 2006
//  S_Decode  <=  '1'  when  (T_IOCS_i  =  '0'  and  S_RangeCheck  =  '1'  and
T_CARD_BLE0_i='0')

    assign S_Decode = ((~T_IOCS_i) & S_RangeCheck);


// Bidirctional Data IO
//
//S_IOTemp either status data or experiment data (from FIFO),
// set in read process

//     assign T_DATA_io = (S_Decode & (~T_IOREAD_i)) ? S_IOTemp : 8'bz;

    assign DATA_io_en = (S_Decode & (~T_IOREAD_i));
    assign DATA_io_out = S_IOTemp;


//MLS this should work for new board (right now they don't overlap right)
//iow <= '0' when (T_IOCS_i = '0' and T_IOWRITE_i = '0') else '1';
//ior <= '0' when (T_IOCS_i = '0' and T_IOREAD_i = '0') else '1';



        always @(posedge clk) begin
                iow_d <= T_IOWRITE_i;
        end

        always @(posedge clk or posedge wr_reset) begin
                        if(wr_reset) begin
                                got_data <= 0;
                        end
                        else begin
// New signal T_CARD_BLE0_i added 7 Jun 2006
//         if (s_decode = '1' and T_CARD_BLE0_i='0' and T_IOWRITE_i = '0' and
iow_d = '1') then
                                if(S_Decode & (~T_IOWRITE_i) & iow_d) begin
                                        got_data <=1;
                                        s_data_i <=T_DATA_io;
                                        s_address_i <= T_ADDRESS_i;
                                end
                        end
        end

        always @(posedge clk or posedge rst) begin
                if (rst) begin
                        got_data_reg <= 0;
                        wr_reset <= 1;
                end
                else begin
                        got_data_reg <= got_data;
                        wr_reset <= got_data_reg;
                end
        end
```

```
// ARM Processor Writes to PC/104
            always @(posedge clk or posedge rst) begin
                  if (rst) begin
                        data_ack <=0;
                        data_rdy <=0;

                        intrpt_ack <=0;
                        intrpt_on <=0;
                        intrpt_off <= 0;
                  end
                  else begin
                        data_ack <= DATA_ACK_i;

                        if (data_ack) begin
                              data_rdy <= 0;
                        end

                        intrpt_ack <= 0;
                        intrpt_on <= 0;
                        intrpt_off <= 0;

                        if(got_data) begin
                              case (s_address_i)
                              `C_DATA_ADDR:
                                    begin
                                          DATA_o[7:0] <= s_data_i; // will
be the XSOC data bus
                                          data_rdy <= 1;
                                    end
                              `C_INTACK_ADDR:
                                    intrpt_ack <= 1;
                              `C_INTON_ADDR:
                                    intrpt_on <= 1;
                              `C_INTOFF_ADDR:
                                    intrpt_off <= 1;

                              endcase
                        end
                  end
            end

      // ARM Processor Reads from PC/104
      always @(T_ADDRESS_i, DATA_i_reg,status_reg_iord) begin
            case (T_ADDRESS_i)
                  `C_DATA_ADDR:
                        begin
                              fifo_rd <= 1;
                              S_IOTemp <= DATA_i_reg;
                        end
                  `C_STATUS_ADDR:
                        begin
                              fifo_rd <= 0;
                              S_IOTemp <=status_reg_iord;      //processor
read status reg (tells whether ready for write/read)

                  //status_reg_iord(0) = not fifo_empty (ie processor can read
a byte)

                  //status_reg_iord(1) = 1 when board is busy reading a byte
(proc can't write yet)
```

```
                    //0 when board is ready for processor to write next byte

                    //status_reg_iord(2) = fifo_almost_full (not used...)
                           end
                    default :
                           begin
                                  fifo_rd <= 0;
                                  S_IOTemp <= 0;
                           end
             endcase
      end

      always @(posedge clk or posedge rst) begin
             if (rst) begin
                    T_INTRPT_o <= 1;
                    intrpt_active <=1;
             end
             else begin
                    intrpt_cnt <= intrpt_cnt + 1;

                    if (intrpt_off)
                           intrpt_active <= 0;
                    else if (intrpt_on)
                           intrpt_active <= 1;

                    if (intrpt_ack)
                           T_INTRPT_o <= 1;
                    else if (fifo_almost_full & intrpt_active)
                           T_INTRPT_o <= 0;
             end
      end

      assign status_reg_iord[7:3] = 5'b0;
      assign status_reg_iord[`STATUS_REG_ALLOW_PROC_READ] = ~fifo_empty;
             //if data in fifo, tell processor it can read a byte
      //(status_reg_iord(0) = '1' for ok to read)
      assign   status_reg_iord[`STATUS_REG_BLOCK_PROC_WRITE]   =   (got_data   |
data_rdy);
             //if FPGA is still processing previous byte from PC/104,
      //tell proc we don't want any more data yet
      //(status_reg_iord(1) = '1' for not ok to write)
      assign status_reg_iord[`STATUS_REG_FIFO_ALMOST_FULL] = fifo_almost_full;

      assign DATA_RDY_o = (data_rdy & (~DATA_ACK_i) & (~data_ack));

      assign fifo_rd_clk = T_IOREAD_i;  //changed since active low

      // assign fifo_din[15:8] = 8'b0;
      assign fifo_din[7:0] = DATA_i;
      assign DATA_i_reg = fifo_dout[7:0];
      assign FIFO_FULL_o = DATA_WREN_i ? 1'b1 : fifo_almost_full;
      assign fifo_wren = DATA_WREN_i;
      assign fifo_ainit = 1'b0;

endmodule




module xsoc_pc104(clk, rst, ctrl, int_req,  sel, d, T_DATA_io,
```

```
            T_ADDRESS_i, T_IOREAD_i, T_IOWRITE_i, T_IOCS_i, T_INTRPT_o,
            fifo_full, errorbus);
    input                 clk;
    input                 rst;
    input [47:0]    ctrl;
        output [2:0] int_req;
    input [2:0]     sel;
    inout [47:0]    d;
    inout [7:0]     T_DATA_io;
    input [9:0]     T_ADDRESS_i;
    input                 T_IOREAD_i;
    input                 T_IOWRITE_i;
    input                 T_IOCS_i;
    output                T_INTRPT_o;
        output                    fifo_full;
        inout [15:0] errorbus;

        wire              clk;
    wire                  rst;
    wire [47:0]     ctrl;
    wire [2:0]      sel;
    tri      [47:0]       d;
        wire  [2:0]       int_req;
    tri      [7:0]  T_DATA_io;
    wire [9:0]      T_ADDRESS_i;
    wire                  T_IOREAD_i;
    wire                  T_IOWRITE_i;
    wire                  T_IOCS_i;
        wor [15:0]        errorbus;

            // fifo signals
            wire   [2:0]  fifo_rd_clk;
            wire   [2:0] fifo_rd;
            wire                fifo_full;
            wire                fifo_almost_full;
            wire                fifo_empty;
            wire [23:0] fifo_din;
            wire [7:0]  fifo_dout;
            wire [2:0]   fifo_wren;

            // XSOC Interface signals
            wire [14:0]  addr;
            wire [2:0]   ud_t, ld_t, ud_ce, ld_ce;

            wire [47:0] dout;


            ctrl_dec_syn  dec0(.clk(clk),  .rst(rst),  .ctrl(ctrl),  .sel(sel),
.ud_t(ud_t[0]),
                         .ld_t(ld_t[0]),  .ud_ce(ud_ce[0]),  .ld_ce(ld_ce[0]),
.addr(addr[4:0]),
                         .errorbus(errorbus), .id(`PC104_CTRL_DEC));

            ctrl_dec_syn  dec1(.clk(clk),  .rst(rst),  .ctrl(ctrl),  .sel(sel),
.ud_t(ud_t[1]),
                         .ld_t(ld_t[1]),  .ud_ce(ud_ce[1]),  .ld_ce(ld_ce[1]),
.addr(addr[9:5]),
                         .errorbus(errorbus), .id(`PC104_CTRL_DEC));


            ctrl_dec_syn  dec2(.clk(clk),  .rst(rst),  .ctrl(ctrl),  .sel(sel),
.ud_t(ud_t[2]),
```

```
                            .ld_t(ld_t[2]),  .ud_ce(ud_ce[2]),  .ld_ce(ld_ce[2]),
.addr(addr[14:10]),
                             .errorbus(errorbus), .id(`PC104_CTRL_DEC));


             assign d[7:0] = ld_t[0] ? 8'bz : dout[7:0];
             assign d[15:8] = ud_t[0] ? 8'bz : dout[15:8];
             assign d[23:16] = ld_t[1] ? 8'bz : dout[23:16];
             assign d[31:24] = ud_t[1] ? 8'bz : dout[31:24];
             assign d[39:32] = ld_t[2] ? 8'bz : dout[39:32];
             assign d[47:40] = ud_t[2] ? 8'bz : dout[47:40];


             wire [23:0]         Data_io;
             wire [2:0]          Data_io_en;
             wire [7:0]          v_Data_io;
             wire                       v_Data_io_en;
             wire [2:0]          INTRPT_o;

             voter1n #(8) Data_io_v (.din(Data_io), .dout(v_Data_io));
             voter1n #(1) Data_io_en_v (.din(Data_io_en), .dout(v_Data_io_en));
             voter1n #(1) INTRPT_o_v (.din(INTRPT_o), .dout(T_INTRPT_o));

             assign T_DATA_io = v_Data_io_en ? v_Data_io[7:0] : 8'bz;

             xsoc_pc104_mod        xsoc_pc104_mod0(.clk(clk),        .rst(rst),
.int_req(int_req[0]),
                                  .d(d), .T_DATA_io(T_DATA_io),
                                  .DATA_io_out(Data_io[7:0]),
.DATA_io_en(Data_io_en[0]),
                                  .T_ADDRESS_i(T_ADDRESS_i),
                                  .T_IOREAD_i(T_IOREAD_i),
.T_IOWRITE_i(T_IOWRITE_i),
                                  .T_IOCS_i(T_IOCS_i), .T_INTRPT_o(INTRPT_o[0]),
                                  .fifo_full(fifo_full),      .dout(dout[15:0]),
.fifo_din(fifo_din[7:0]),
                                  .fifo_wren(fifo_wren[0]),
.fifo_rd(fifo_rd[0]),
                                  .fifo_rd_clk(fifo_rd_clk[0]),
.fifo_dout(fifo_dout[7:0]),
                                  .fifo_empty(fifo_empty),
.fifo_almost_full(fifo_almost_full),
                                  .ld_t(ld_t), .ld_ce(ld_ce), .addr(addr),
                                  .errorbus(errorbus));

             xsoc_pc104_mod        xsoc_pc104_mod1(.clk(clk),        .rst(rst),
.int_req(int_req[1]),
                                  .d(d), .T_DATA_io(T_DATA_io),
                                  .DATA_io_out(Data_io[15:8]),
.DATA_io_en(Data_io_en[1]),
                                  .T_ADDRESS_i(T_ADDRESS_i),
                                  .T_IOREAD_i(T_IOREAD_i),
.T_IOWRITE_i(T_IOWRITE_i),
                                  .T_IOCS_i(T_IOCS_i), .T_INTRPT_o(INTRPT_o[1]),
                                  .fifo_full(fifo_full),     .dout(dout[31:16]),
.fifo_din(fifo_din[15:8]),
                                  .fifo_wren(fifo_wren[1]),
.fifo_rd(fifo_rd[1]),
                                  .fifo_rd_clk(fifo_rd_clk[1]),
.fifo_dout(fifo_dout[7:0]),
                                  .fifo_empty(fifo_empty),
.fifo_almost_full(fifo_almost_full),
```

```
                                 .ld_t(ld_t), .ld_ce(ld_ce), .addr(addr),
                                 .errorbus(errorbus));

             xsoc_pc104_mod        xsoc_pc104_mod2(.clk(clk),        .rst(rst),
.int_req(int_req[2]),
                                 .d(d), .T_DATA_io(T_DATA_io),
                                 .DATA_io_out(Data_io[23:16]),
.DATA_io_en(Data_io_en[2]),
                                 .T_ADDRESS_i(T_ADDRESS_i),
                                 .T_IOREAD_i(T_IOREAD_i),
.T_IOWRITE_i(T_IOWRITE_i),
                                 .T_IOCS_i(T_IOCS_i), .T_INTRPT_o(INTRPT_o[2]),
                                 .fifo_full(fifo_full),     .dout(dout[47:32]),
.fifo_din(fifo_din[23:16]),
                                 .fifo_wren(fifo_wren[2]),
.fifo_rd(fifo_rd[2]),
                                 .fifo_rd_clk(fifo_rd_clk[2]),
.fifo_dout(fifo_dout[7:0]),
                                 .fifo_empty(fifo_empty),
.fifo_almost_full(fifo_almost_full),
                                 .ld_t(ld_t), .ld_ce(ld_ce), .addr(addr),
                                 .errorbus(errorbus));

             pc104queue_8x16 pc104_queue1(.din(fifo_din),
                                                                  .clk(clk),
                                                                  .rst(rst),

       .add(fifo_wren),

       .remove(fifo_rd),
                                                                  //  since  the
rd clock is an input to the FPGA voting doesn't help

       .remove_clk(T_IOREAD_i),

       .dout(fifo_dout[7:0]),

       .empty(fifo_empty),

       .nearlyfull(fifo_almost_full),

       .full(fifo_full));

// comment out the above module instantiation and uncomment this one
// to switch back to the CoreGen queue for testing
//fifo_out_arm fifo_out_arm1 (
//                  .din(fifo_din[7:0]),
//                  .wr_en(fifo_wren[0]),
//                  .wr_clk(clk),
//                  .rd_en(fifo_rd[0]),
//                  .rd_clk(fifo_rd_clk[0]),
//                  .dout(fifo_dout[7:0]),
//                  .empty(fifo_empty),
//                  .full(fifo_full),
//                  .almost_full(fifo_almost_full));

endmodule
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: VHDL SOURCE CODE

This appendix contains source code for the two VHDL modules from the previous X1 configuration modified for use with the CFTP XSOC microcontroller.

A. SELECTMAP_CONFIG.VHD

```
--------------------------------------------------------------------------------
-- selectmap_config.vhd                                                       --
-- Author: Mindy Surratt, 2005                                                --
-- Research Associate, Naval Postgraduate School, Monterey, CA                --
--                                                                            --
-- Code to perform a selectmap full configuration on the V1 experiment FPGA   --
--------------------------------------------------------------------------------

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 selectmap_config is

    port (

        T_CLOCK_i           : in std_logic;
        RESET_i             : in std_logic;

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

        T_SELECTMAP_INIT_o  : out std_logic;
        T_SELECTMAP_WRITE_o : out std_logic;
        T_SELECTMAP_CS_o    : out std_logic;
        T_SELECTMAP_DATA_o  : out std_logic_vector(7 downto 0);

        SM_CONFIG_RQST_i    : in std_logic;
        SM_CONFIG_STATUS_o  : out std_logic;
               SM_FLASH_BASE_i           : in std_logic_vector(20 downto 0);

        T_FLASH_DATA_i      : in std_logic_vector(15 downto 0);
        T_FLASH_ADDRESS_o   : out std_logic_vector(21 downto 0)

--        PC104_WR_RDY_i      : in std_logic;
--        PC104_WR_EN_o       : out std_logic;
--        DATA_o              : out std_logic_vector(7 downto 0)

    );

end selectmap_config;

architecture rtl of selectmap_config is

CONSTANT BIN_LENGTH             : integer := 450996; --SIMULATION 20;
--CONSTANT CONFIG_DELAY           : integer := 40; --SIMULATION 10;
--Added 20 clocks to config delay to offset deletion of PC/104 data output
CONSTANT CONFIG_DELAY           : integer := 60; --SIMULATION 10;
```

```
CONSTANT ABORT_SETUP_LENGTH     : integer := 5;
CONSTANT ABORT_LENGTH           : integer := 5;
CONSTANT ABORT_RELEASE_LENGTH   : integer := 5;
CONSTANT CONFIG_LENGTH          : integer := CONFIG_DELAY + BIN_LENGTH +
                                                ABORT_SETUP_LENGTH +
                                                ABORT_LENGTH +
                                                ABORT_RELEASE_LENGTH;

signal count_config         : integer range 0 to CONFIG_LENGTH ;
signal byte_count           : integer range 0 to 100;
-- signal wr_cnt               : integer range 0 to 100;
CONSTANT SC_WORD_LENGTH     : integer := 10;
--type    sc_vector is array(SC_WORD_LENGTH-1 downto 0) of std_logic_vector(7
downto 0);
--signal sc_word               : sc_vector;
signal s_flash_address_o    : std_logic_vector(20 downto 0);

begin

-- only one flash device on CFTP-1, so only 21 addr lines (addr(21) = '0')
T_FLASH_ADDRESS_o(21) <= '0';
T_FLASH_ADDRESS_o(20 downto 0) <= s_flash_address_o;


-- Selectmap Configuration Process
process(T_CLOCK_i, RESET_i)
begin
       if (RESET_i = '1') then

        s_flash_address_o <= "000000000000000000000";

        SM_CONFIG_STATUS_o <= '0';

        T_SELECTMAP_DATA_o <= x"00";

              T_SELECTMAP_INIT_o <= '0';
              T_SELECTMAP_WRITE_o <= '1';
              T_SELECTMAP_CS_o <= '1';

--     PC104_WR_EN_o <= '0';

--     wr_cnt <= 0;

              count_config <= CONFIG_LENGTH;

       elsif(T_CLOCK_i'event and T_CLOCK_i = '1') then

        T_SELECTMAP_INIT_o <= '1';
        T_SELECTMAP_CS_o <= '1';
        T_SELECTMAP_WRITE_o <= '1';
   --   PC104_WR_EN_o <= '0';

        if ( count_config = 0 ) then
       --                  if (wr_cnt = 0) then
   --            sc_word(0) <= x"53";
   --            sc_word(1) <= x"43";
   --            sc_word(2) <= TIMESTAMP_i(63 downto 56);
   --            sc_word(3) <= TIMESTAMP_i(55 downto 48);
   --            sc_word(4) <= TIMESTAMP_i(47 downto 40);
   --            sc_word(5) <= TIMESTAMP_i(39 downto 32);
   --            sc_word(6) <= TIMESTAMP_i(31 downto 24);
   --             sc_word(7) <= TIMESTAMP_i(23 downto 16);
```

```
   --              sc_word(8) <= TIMESTAMP_i(15 downto 8);
   --              sc_word(9) <= TIMESTAMP_i(7 downto 0);
   --              wr_cnt <= wr_cnt + 1;
   --          elsif (wr_cnt < SC_WORD_LENGTH+1 and PC104_WR_RDY_i = '1') then
   --              DATA_o <= sc_word(wr_cnt-1);
   --              PC104_WR_EN_o <= '1';
   --              wr_cnt <= wr_cnt + 1;
   --          elsif (wr_cnt = SC_WORD_LENGTH+1) then
                 count_config <= 1;
   --              wr_cnt <= 0;
   --          end if;

        -- give it some time
        elsif (count_config < CONFIG_DELAY ) then
            count_config <= count_config + 1;

        -- assert write and CS
        elsif (count_config < CONFIG_DELAY + ABORT_SETUP_LENGTH) then
            count_config <= count_config + 1;
            T_SELECTMAP_CS_o <= '0';
            T_SELECTMAP_WRITE_o <= '0';

        -- deassert write while CS is asserted (pulls an abort)
        elsif (count_config < CONFIG_DELAY + ABORT_SETUP_LENGTH + ABORT_LENGTH)
then
            count_config <= count_config + 1;
            T_SELECTMAP_CS_o <= '0';
            T_SELECTMAP_WRITE_o <= '1';

        -- Just deassert WRITE and CS to release the abort (for Virtex 1)
        -- (Default values of WRITE and CS are '1', see above)
        elsif (count_config < CONFIG_DELAY + ABORT_SETUP_LENGTH +
                                ABORT_LENGTH + ABORT_RELEASE_LENGTH) then
            count_config <= count_config + 1;

        -- read data from flash and write to X2s selectmap interface
        elsif (count_config < CONFIG_LENGTH) then
            byte_count <= byte_count + 1;

            --T_SELECTMAP_DATA_io0 is the MSB (except I swapped values in UCF
            --so in this instance D7 is MSB...)
            -- disabled bit flipping in promgen (default is to flip bits in
            -- each byte), so T_s_o<=t_flash_data instead of
            -- having to reverse the bits (7=0, 6=1, etc)

            -- takes some time to read from the flash (possibly less than
            -- I'm allowing for...
            if (byte_count = 6) then

--***
--***                    byte_count <= 6;
--***
--***            if (wr_cnt = 0) then
--***                sc_word(0) <= T_FLASH_DATA_i(7 downto 0);
--***                wr_cnt <= wr_cnt + 1;
--***            elsif (wr_cnt < 2 and PC104_WR_RDY_i = '1') then
--***                DATA_o <= sc_word(wr_cnt-1);
--***                PC104_WR_EN_o <= '1';
--***                wr_cnt <= wr_cnt + 1;
--***            elsif (wr_cnt = 2) then
--***
```

```
                T_SELECTMAP_CS_o <= '0';
                T_SELECTMAP_WRITE_o <= '0';
                T_SELECTMAP_DATA_o <= T_FLASH_DATA_i(7 downto 0);
                count_config <= count_config + 1;
                byte_count <= 0;
                s_flash_address_o <= s_flash_address_o + 1;

--***
--***                wr_cnt <= 0;
--***            end if;
--***

            end if;

        -- sit here until we get a config request from the top level
        elsif (count_config = CONFIG_LENGTH ) then
            count_config <= CONFIG_LENGTH;
            SM_CONFIG_STATUS_o <= '0';
            --reset flash address
 --           s_flash_address_o <= "000000000000000000000";
 --                 08202008 added selectable base address ded
                            s_flash_address_o <= SM_FLASH_BASE_i(20 downto 0);
            if (SM_CONFIG_RQST_i = '1') then
                count_config <= 0;
               SM_CONFIG_STATUS_o <= '1';
            end if;

        end if;


    end if;

end process;

end rtl;
```

B. SELECTMAP_READBACK.VHD

```
--------------------------------------------------------------------------------
-- selectmap_readback.vhd                                                     --
-- Author: Mindy Surratt, 2005                                                --
-- Research Associate, Naval Postgraduate School, Monterey, CA                --
--                                                                            --
-- Code to perform a selectmap readback on the experiment FPGA,               --
-- compare the readback data with the configuration file and mask file stored --
-- in the Flash memory, and output any configuration errors to the PC/104     --
--------------------------------------------------------------------------------

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 selectmap_readback is
    port (

        T_CLOCK_i           : in std_logic;
        CLOCK_i             : in std_logic;
```

```
 --       CLOCK_NOBUFG_i      : in std_logic;
        RESET_i             : in std_logic;

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

        T_CCLK_o            : out std_logic;
               T_SELECTMAP_FLASH_BASE_i :      in std_logic_vector(20 downto 0);
        T_SELECTMAP_INIT_o  : out std_logic;
        T_SELECTMAP_WRITE_o : out std_logic;
        T_SELECTMAP_CS_o    : out std_logic;
        T_SELECTMAP_DATA_o  : out std_logic_vector(7 downto 0);
        T_SELECTMAP_DATA_i  : in std_logic_vector(7 downto 0);

        SM_RB_RQST_i        : in std_logic;
        SM_RB_STATUS_o      : out std_logic;
               SM_RB_ACK_i               : in std_logic;

        T_FLASH_DATA_i      : in std_logic_vector(15 downto 0);
        T_FLASH_ADDRESS_o   : out std_logic_vector(21 downto 0);

--        PC104_WR_RDY_i      : in std_logic;
--        PC104_WR_EN_o       : out std_logic;
--        DATA_o              : out std_logic_vector(7 downto 0)
             SM_RB_READ_DATA_o   : out std_logic_vector (7 downto 0);
             SM_RB_ERROR_LOC_H_o : out std_logic_vector (7 downto 0);
             SM_RB_ERROR_LOC_L_o : out std_logic_vector (15 downto 0);
             SM_RB_ERROR_WORD_o  : out std_logic_vector (15 downto 0);
             SM_RB_DATA_RDY_o           : out std_logic
             );
end selectmap_readback;

architecture rtl of selectmap_readback is

-- Constants

    CONSTANT READBACK_COMMAND_LENGTH    : integer := 36;
    -- CLB_LENGTH refers to CLBs, IOBs, and BRAMIs. We do not read back BRAMs
    CONSTANT CLB_LENGTH                 : integer := 435240; --SIMULATION 130;
    CONSTANT READBACK_DELAY             : integer := 40; --SIMULATION 10;
    CONSTANT READBACK_INIT1_DELAY       : integer := 1;
    CONSTANT READBACK_INIT2_DELAY       : integer := 1; -- DO NOT CHANGE THIS!
    -- Number of clocks needed to set up readback (before we actually start
    -- getting data)
    CONSTANT READBACK_OFFSET            : integer := READBACK_COMMAND_LENGTH +
                                                     READBACK_DELAY +
                                                     READBACK_INIT1_DELAY +
                                                     READBACK_INIT2_DELAY;
    CONSTANT READBACK_LENGTH            : integer := CLB_LENGTH +
                                                     READBACK_OFFSET;
        CONSTANT sleep_st                                     :  std_logic_vector
(2 downto 0) := "000";
        CONSTANT wr_hdr_st                                    :  std_logic_vector
(2 downto 0) := "001";
        CONSTANT read_flash_st                                :  std_logic_vector
(2 downto 0) := "010";
        CONSTANT read_sm_st                                   :  std_logic_vector
(2 downto 0) := "011";
        CONSTANT wait_st                     : std_logic_vector (2 downto 0) :=
"100";
        CONSTANT compare_st                                   :  std_logic_vector
(2 downto 0) := "101";
```

```
        CONSTANT write_pc104_st                         :   std_logic_vector   (2
downto 0) := "110";

        signal s_clock_i                              : std_logic := '0';
        signal report_error_state               : std_logic_vector (2 downto 0);


-- Flash compare/report signals

 --   type   my_state is (sleep_st,read_flash_st,wr_hdr_st,
 --                       read_sm_st, wait_st, compare_st, write_pc104_st);
    type   error_vector is array(15 downto 0) of std_logic_vector(7 downto 0);

--    signal report_error_state       : my_state;
--    signal error_word               : error_vector;

    signal flash_data_reg           : std_logic_vector(15 downto 0);
    signal s_flash_address_o        : std_logic_vector(20 downto 0);
    signal error_count              : integer range 0 to 1024;
    signal error_location           : std_logic_vector (23 downto 0);
    signal error_location_readback  : std_logic_vector (23 downto 0);
 --   signal wr_cnt                   : integer range 0 to 32;

 --   CONSTANT SM_WORD_LENGTH         : integer := 10;
 --   type   sm_vector is array(SM_WORD_LENGTH-1 downto 0) of std_logic_vector(7
downto 0);
 --   signal sm_word               : sm_vector;



-- Selectmap Readback signals

    type   readback_mem_type is array(READBACK_COMMAND_LENGTH-1 downto 0)
                                of std_logic_vector(7 downto 0);

    signal readback_command     : readback_mem_type;
    signal count_readback       : integer range 0 to 1024000;
    signal strt_rb              : std_logic;
    signal s_selectmap_data_d   : std_logic_vector(7 downto 0);
    signal s_selectmap_data_i   : std_logic_vector(7 downto 0);
    signal reading_sm_data      : std_logic;
    signal reading_sm_data_d    : std_logic;
    signal readback_location    : integer range 0 to 1024000;

    signal selectmap_read_data  : std_logic_vector(7 downto 0);


begin

    -- Clocking CCLK only works if we use a signal that is not on the
    -- clock network
    process (T_CLOCK_i) begin
        if (T_CLOCK_i'event and T_CLOCK_i = '1') then
            s_clock_i <= not s_clock_i;
        end if;
    end process;


    -- Command sequence that initializes a selectmap readback
    readback_command(0)(7 downto 0) <= x"FF"; --MSB dummy word
    readback_command(1)(7 downto 0) <= x"FF";
    readback_command(2)(7 downto 0) <= x"FF";
```

```
  readback_command(3)(7 downto 0) <= x"FF";
  readback_command(4)(7 downto 0) <= x"AA"; --MSB synchronization word
  readback_command(5)(7 downto 0) <= x"99";
  readback_command(6)(7 downto 0) <= x"55";
  readback_command(7)(7 downto 0) <= x"66";
  readback_command(8)(7 downto 0) <= x"30"; --MSB write 1 word to FAR reg
  readback_command(9)(7 downto 0) <= x"00";
  readback_command(10)(7 downto 0) <= x"20";
  readback_command(11)(7 downto 0) <= x"01";
  readback_command(12)(7 downto 0) <= x"00"; --MSB FAR address
  readback_command(13)(7 downto 0) <= x"00";
  readback_command(14)(7 downto 0) <= x"00";
  readback_command(15)(7 downto 0) <= x"00";
  readback_command(16)(7 downto 0) <= x"30"; --MSB write 1 word to CMD reg
  readback_command(17)(7 downto 0) <= x"00";
  readback_command(18)(7 downto 0) <= x"80";
  readback_command(19)(7 downto 0) <= x"01";
  readback_command(20)(7 downto 0) <= x"00"; --MSB RCFG command
  readback_command(21)(7 downto 0) <= x"00";
  readback_command(22)(7 downto 0) <= x"00";
  readback_command(23)(7 downto 0) <= x"04";
  readback_command(24)(7 downto 0) <= x"28"; --MSB type 2 header
  readback_command(25)(7 downto 0) <= x"00";
  readback_command(26)(7 downto 0) <= x"60";
  readback_command(27)(7 downto 0) <= x"00";
  readback_command(28)(7 downto 0) <= x"48"; --type 2 read from active reg
  readback_command(29)(7 downto 0) <= x"01"; --(FDRO) 0x1A90A 32 bit words
  readback_command(30)(7 downto 0) <= x"A9"; -- (READBACK_LENGTH/4)
  readback_command(31)(7 downto 0) <= x"0A";
  readback_command(32)(7 downto 0) <= x"00"; --MSB pad word
  readback_command(33)(7 downto 0) <= x"00";
  readback_command(34)(7 downto 0) <= x"00";
  readback_command(35)(7 downto 0) <= x"00";


  -- grabs selectmap data on the next read_sm_st,
  -- the first byte read back will be junk (no data yet)
  T_CCLK_o            <= '1' when reading_sm_data = '1'
                                 and report_error_state /= read_sm_st
                            else s_clock_i;
  s_selectmap_data_i  <= T_SELECTMAP_DATA_i;

  -- only one flash device on CFTP-1, so only 21 addr lines (addr(21) = '0')
  T_FLASH_ADDRESS_o(21) <= '0';
  T_FLASH_ADDRESS_o(20 downto 0) <= s_flash_address_o;

  readback_location <= 0  when count_readback < READBACK_OFFSET
                          else (count_readback - READBACK_OFFSET);


  process(CLOCK_i,RESET_i)
  begin
      if (RESET_i = '1') then

          report_error_state <= sleep_st;
          error_count <= 0;
--s          wr_cnt <= 0;

--          PC104_WR_EN_o <= '0';
--          DATA_o <= x"31";

      elsif (CLOCK_i'event and CLOCK_i = '1') then
```

```
--          PC104_WR_EN_o <= '0';
            report_error_state <= report_error_state;

          case report_error_state is

            when sleep_st =>

                -- set flash address to just after the selectmap
                -- configuration commands in the bin file
 --              s_flash_address_o <= "00000000000001001000";
                                                 s_flash_address_o                <=
T_SELECTMAP_FLASH_BASE_i;
                -- wait until we start actually reading SM data
                -- then start the readback reports
                if (reading_sm_data = '1') then
                    report_error_state <= wr_hdr_st;
                end if;

            when wr_hdr_st =>

  --              if (wr_cnt = 0) then
  --                  sm_word(0) <= x"53";
  --                 sm_word(1) <= x"4D";
  --                  sm_word(2) <= TIMESTAMP_i(63 downto 56);
  --                  sm_word(3) <= TIMESTAMP_i(55 downto 48);
  --                  sm_word(4) <= TIMESTAMP_i(47 downto 40);
  --                 sm_word(5) <= TIMESTAMP_i(39 downto 32);
  --                  sm_word(6) <= TIMESTAMP_i(31 downto 24);
  --                  sm_word(7) <= TIMESTAMP_i(23 downto 16);
  --                  sm_word(8) <= TIMESTAMP_i(15 downto 8);
  --                  sm_word(9) <= TIMESTAMP_i(7 downto 0);
  --                  wr_cnt <= wr_cnt + 1;
  --              elsif (wr_cnt < SM_WORD_LENGTH+1 and PC104_WR_RDY_i = '1')
then
  --                  DATA_o <= sm_word(wr_cnt-1);
  --                  PC104_WR_EN_o <= '1';
  --                  wr_cnt <= wr_cnt + 1;
  --                  report_error_state <= wr_hdr_st;
  --              elsif (wr_cnt = SM_WORD_LENGTH+1) then
  --                 wr_cnt <= 0;
                   report_error_state <= read_flash_st;
  --              end if;

            when read_flash_st =>

                report_error_state <= read_sm_st;

                -- Go idle if we are done reading back
                if (reading_sm_data = '0') then
                    report_error_state <= sleep_st;
                end if;

            when read_sm_st =>

                -- latch the selectmap data from the last read,
                -- and trigger CCLK to get the next byte (see T_CCLK_o
                -- assignment above)
                selectmap_read_data <= s_selectmap_data_i;
                report_error_state <= wait_st;

            when wait_st =>
```

```
                    -- Skip first junk byte (see CCLK above), 1 dummy word,
                    -- and one pad frame (120 bytes)
                    if (readback_location < 125) then
                        report_error_state <= read_sm_st;
                    else
                        report_error_state <= compare_st;
                    end if;

                when compare_st =>

                    -- latch flash data for possible report to PC104
                    flash_data_reg <= T_FLASH_DATA_i;
                    -- increment the flash address for the next read
                    s_flash_address_o <= s_flash_address_o + 1;

                    -- Compare read back selectmap data, configuration data
                    -- stored in the flash, and mask data stored in the flash.
                    -- If there is a mismatch, print a report to PC/104.
                    -- Otherwise, read the next byte
                    if ( ( (T_FLASH_DATA_i(7 downto 0) xor selectmap_read_data)
                        and not T_FLASH_DATA_i(15 downto 8) ) /= x"00" ) then
                        report_error_state <= write_pc104_st;
                    else
                        report_error_state <= read_flash_st;
                    end if;

                when write_pc104_st =>

         -- 08/01/2008 ded comment out direct access to PC/104 bus and add ack
from CPU
                    -- print out SM report
                    -- error_word() defined below
                                                 SM_RB_DATA_RDY_o <= '1';
  --                  if (wr_cnt < 7 and PC104_WR_RDY_i = '1' ) then
  --                      DATA_o <= error_word(wr_cnt);
  --                      PC104_WR_EN_o <= '1';
  --                      wr_cnt <= wr_cnt + 1;
                    -- read next byte from selectmap when done
  --                  elsif (wr_cnt = 7) then
                                                  if (SM_RB_ACK_i = '1') then
                                                              report_error_state
<= read_flash_st;
                                                              SM_RB_DATA_RDY_o  <=
'0';
                                                  end if;
  --                      wr_cnt <= 0;
  --                  end if;

                when others => report_error_state <= sleep_st;
            end case;
        end if;

    end process;

    error_location <= "000" & (s_flash_address_o - 1) ;
    error_location_readback <=
            std_logic_vector(to_unsigned(readback_location,24));

    -- SMRB error report, error location is byte location within
    -- bin file
--    error_word(0) <= x"00";
```

```
--    error_word(1) <= selectmap_read_data ;
--    error_word(2) <= error_location(23 downto 16);
--    error_word(3) <= error_location(15 downto 8);
--    error_word(4) <= error_location(7 downto 0);
--    error_word(5) <= flash_data_reg(7 downto 0);
--    error_word(6) <= flash_data_reg(15 downto 8);
    SM_RB_READ_DATA_o <= selectmap_read_data ;
    SM_RB_ERROR_LOC_H_o <= error_location(23 downto 16);
    SM_RB_ERROR_LOC_L_o <= error_location(15 downto 0);
    SM_RB_ERROR_WORD_o <= flash_data_reg(15 downto 0);




    -- Process to perform Selectmap Readback
    process(CLOCK_i,RESET_i)
    begin
        if (RESET_i = '1') then

            T_SELECTMAP_INIT_o <= '1';
            T_SELECTMAP_WRITE_o <= '1';
            T_SELECTMAP_CS_o <= '1';

            count_readback <= READBACK_LENGTH;
            SM_RB_STATUS_o <= '0';
            reading_sm_data <= '0';
            reading_sm_data_d <= '0';

            s_selectmap_data_d <= x"FF";

            strt_rb <= '0';

            T_SELECTMAP_DATA_o <= x"FF";

        elsif(CLOCK_i'event and CLOCK_i = '1') then

            T_SELECTMAP_INIT_o <= '1';
            T_SELECTMAP_CS_o <= '1';
            T_SELECTMAP_WRITE_o <= '1';

            reading_sm_data <= '0';
            reading_sm_data_d <= reading_sm_data;

            -- Give it some time
            if (count_readback < READBACK_DELAY) then
                count_readback <= count_readback + 1;

            -- Write the readback commands to X2s selectmap interface
            elsif (count_readback < READBACK_COMMAND_LENGTH +
                                    READBACK_DELAY ) then
                T_SELECTMAP_CS_o <= '0';
                T_SELECTMAP_WRITE_o <= '0';
                T_SELECTMAP_DATA_o <=
                readback_command(count_readback - READBACK_DELAY );
                count_readback <= count_readback + 1;

            --INITialize readback (deassert CS and WRITE)
            elsif (count_readback < READBACK_COMMAND_LENGTH + READBACK_DELAY +
                                    READBACK_INIT1_DELAY ) then
                T_SELECTMAP_CS_o <= '1';
                T_SELECTMAP_WRITE_o <= '1';
```

```
                count_readback <= count_readback + 1;

            -- INITialize readback (assert CS)
            elsif (count_readback < READBACK_OFFSET ) then
                T_SELECTMAP_CS_o <= '0';
                T_SELECTMAP_WRITE_o <= '1';
                count_readback <= count_readback + 1;

            -- read back the configuration data from X2
            elsif ((count_readback < (READBACK_LENGTH)) ) then

                T_SELECTMAP_CS_o <= '0';
                T_SELECTMAP_WRITE_o <= '1';
                reading_sm_data <= '1';
                -- data_d will be FF until second clock in this statement
                -- this prevents SM readback from hanging on the next statement
                -- if there happen to be no FFs at the beginning
                s_selectmap_data_d <= s_selectmap_data_i;

                -- Skip all dummy FF data coming out of the selectmap interface
                -- at the beginning
                if (s_selectmap_data_i /= x"FF" and
                    s_selectmap_data_d = x"FF") then
                    strt_rb <= '1';
                end if;

                -- Don't increment the readback counter unless we've cleared out
                -- all the dummy FF's (strt_rb = '1') and we we are latching
                -- the selectmap data in the read_sm_st of the state machine
                -- CCLK is clocked off of read_sm_st as well, so every time
                -- we are in read_sm_st, another byte of selectmap data is
                -- clocked out, and we grab it the next time we're in read_sm_st
                -- we only want to increment the counter when we actually
                -- grab a selectmap readback byte
                if (report_error_state = read_sm_st and strt_rb = '1' ) then
                    count_readback <= count_readback + 1;
                end if;

            -- sit here until we get a readback request from top level
            elsif (count_readback = READBACK_LENGTH ) then
                count_readback <= READBACK_LENGTH;
                SM_RB_STATUS_o <= '0';
                strt_rb <= '0';
                -- Make sure this starts out as FF so we will start the readback
                -- even if there are no dummy FF's at the beginning of the
                -- readback
                s_selectmap_data_d <= x"FF";

                if (SM_RB_RQST_i = '1') then
                   count_readback <= 0;
                   SM_RB_STATUS_o <= '1';
                end if;

            end if;
        end if;
    end process;

end rtl;
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: C SOURCE CODE

This appendix contains the two C source files used in the CFTP microcontroller. The first, HEXTOV.C, is the source for the .hex to Verilog format conversion tool. The second, libxr16.c, is the runtime library for XSOC.

A. HEXTOV.C

```
#include <stdio.h>
#define  NUMBYTES 8
#define  NUMWORDS 4
#define  NUMOUTPUT 22
#define  HEXPERLINE 64


unsigned char inputbytes[NUMBYTES];
FILE *outputfile;
unsigned char outputhex[NUMOUTPUT][HEXPERLINE];
int linepos;
int linecnt;


void myputc(unsigned char mychar, int mybin);
void writelines();
void clearlines();
unsigned char ascii2bin(int char1, int char2);
unsigned char bin2ascii(unsigned char myint);

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

  FILE *inputfilep;

  char myfilename[15];
  int i,j;
  int tbyte;
  int fileend;
  int char1, char2;
  char inputfilename[255];
  int tlen;


  strcpy(myfilename, argv[1]);
  strcat(myfilename, ".hex");

  if ((inputfilep = fopen(myfilename,"r")) == NULL)
    {
      fprintf(stderr, "Can't open %s\n",myfilename);
      exit(1);
    }


  strcpy(myfilename, argv[1]);
  strcat(myfilename, ".v");
  if ((outputfile=fopen(myfilename,"w"))==NULL)
```

```
  {
    fprintf(stderr, "Can't open %s\n", myfilename);
  }


fileend=0;
linepos=63;
linecnt=0;
clearlines();
while(fileend==0)
  {

    tbyte=getc(inputfilep);
    /*  start a new line */
    if (tbyte!=EOF)
     {
       /* read a byte, read 9 more*/
       for(j=0;j<9;j++)
         {
           getc(inputfilep);
         }
       char1=getc(inputfilep);
       char2=getc(inputfilep);
       inputbytes[1]=ascii2bin(char1,char2);
       getc(inputfilep);
       char1=getc(inputfilep);
       char2=getc(inputfilep);
       inputbytes[0]=ascii2bin(char1,char2);
       getc(inputfilep);
       char1=getc(inputfilep);
       char2=getc(inputfilep);
       inputbytes[3]=ascii2bin(char1,char2);
       getc(inputfilep);
       char1=getc(inputfilep);
       char2=getc(inputfilep);
       inputbytes[2]=ascii2bin(char1,char2);
       getc(inputfilep);
       char1=getc(inputfilep);
       char2=getc(inputfilep);
       inputbytes[5]=ascii2bin(char1,char2);
       getc(inputfilep);
       char1=getc(inputfilep);
       char2=getc(inputfilep);
       inputbytes[4]=ascii2bin(char1,char2);
       getc(inputfilep);
       char1=getc(inputfilep);
       char2=getc(inputfilep);
       inputbytes[7]=ascii2bin(char1,char2);
       getc(inputfilep);
       char1=getc(inputfilep);
       char2=getc(inputfilep);
       inputbytes[6]=ascii2bin(char1,char2);
       getc(inputfilep);

            /*
       if ((linecnt==0) && (linepos==63))
         {
           printf("data is %x %x %x %x %x %x %x %x\n",
                 inputbytes[0], inputbytes[1], inputbytes[2], inputbytes[3],
                 inputbytes[4], inputbytes[5], inputbytes[6], inputbytes[7]);
         }
          */
```

```
    split();


    char1=getc(inputfilep);
    char2=getc(inputfilep);
    inputbytes[1]=ascii2bin(char1,char2);
    getc(inputfilep);
    char1=getc(inputfilep);
    char2=getc(inputfilep);
    inputbytes[0]=ascii2bin(char1,char2);
    getc(inputfilep);
    char1=getc(inputfilep);
    char2=getc(inputfilep);
    inputbytes[3]=ascii2bin(char1,char2);
    getc(inputfilep);
    char1=getc(inputfilep);
    char2=getc(inputfilep);
    inputbytes[2]=ascii2bin(char1,char2);
    getc(inputfilep);
    char1=getc(inputfilep);
    char2=getc(inputfilep);
    inputbytes[5]=ascii2bin(char1,char2);
    getc(inputfilep);
    char1=getc(inputfilep);
    char2=getc(inputfilep);
    inputbytes[4]=ascii2bin(char1,char2);
    getc(inputfilep);
    char1=getc(inputfilep);
    char2=getc(inputfilep);
    inputbytes[7]=ascii2bin(char1,char2);
    getc(inputfilep);
    char1=getc(inputfilep);
    char2=getc(inputfilep);
    inputbytes[6]=ascii2bin(char1,char2);
    getc(inputfilep);
    /*getc(inputfilep); */

    /* if ((linecnt==0) && (linepos==62))
      {
        printf("data is %x %x %x %x %x %x %x %x\n",
              inputbytes[0], inputbytes[1], inputbytes[2], inputbytes[3],
              inputbytes[4], inputbytes[5], inputbytes[6], inputbytes[7]);
      }
  */
    split();

    /*     for(i=1;i<NUMBYTES;i++)
      {
        inputbytes[i]=getc(inputfilep);
        if (inputbytes[i]==EOF)
         {
           inputbytes[i]=0;
           fileend=1;
         }
      }

    */

    }
  else
   {
```

```
        fileend=1;
      }
    }
  /* printf("linepos is %d\n\r",linepos); */
  writelines();

  exit(0);

}

int split()
{
  int i,j;
  unsigned short mywords[NUMWORDS];
  unsigned short mymask;
  unsigned short curchar;
  unsigned short p[22];

  /* change bytes to word */

  for(i=0;i<NUMWORDS;i++)
    {
      mywords[i]=((inputbytes[(1+(i*2))])<<8) | ((inputbytes[(i*2)]));
    }


  mymask=0x1;

  for(i=0;i<16;i++)
    {
      curchar=0;
      for(j=3;j>(-1);j--)
       {
         if((mywords[j] & mymask)!=0)
           {
             curchar = curchar | 0x1;
           }
         if (j>0)
           {
             curchar=curchar << 1;
           }

       }

      if(curchar < 10)
       {
         curchar = curchar + 0x30;
       }
      else
       {
         curchar = curchar + 0x41 - 10;
       }

      myputc(curchar,i);

      mymask=mymask << 1;
    }

  for(i=16;i<22;i++)
    {
      p[i]=0;
    }
```

```
for(i=3;i>(-1);i--)
  {

    mymask=0x1;

    for(j=0;j<16;j++)
     {
       if((mymask & mywords[i])!=0)
         {
           if((j==0) || (j==1) || (j==3) || (j==4) || (j==6) || (j==8)
             || (j==10) || (j==11) || (j==13) || (j==15))
            {
              p[16]=p[16] ^ (0x1);
            }
           if((j==0) || (j==2) || (j==3) || (j==5) || (j==6) || (j==9)
             || (j==10) || (j==12) || (j==13))
            {
              p[17]=p[17] ^ (0x1);
            }
           if((j==1) || (j==2) || (j==3) || (j==7) || (j==8) || (j==9)
             || (j==10) || (j==14) || (j==15))
            {
              p[18]=p[18] ^ (0x1);
            }
           if((j==4) || (j==5) || (j==6) || (j==7) || (j==8) || (j==9)
             || (j==10))
            {
              p[19]=p[19] ^ (0x1);
            }
           if((j==11) || (j==12) || (j==13) || (j==14) || (j==15))
            {
              p[20]=p[20] ^ (0x1);
            }
           p[21]=p[21] ^ (0x1);
         }
       mymask=mymask << 1;
     }

    mymask=0x1;

    for(j=16;j<21;j++)
     {
       if((p[j] & mymask) !=0)
         {
           p[21]=p[21] ^ (0x1);
         }
     }

    if(i>0) {
    for(j=16;j<22;j++)
     {
       p[j]=p[j] << 1;
     }
    }
  }
for(i=16;i<22;i++)
  {
    myputc(bin2ascii(p[i]),i);
  }
linepos--;
```

```
}

void myputc(unsigned char mychar, int mybin)
{
  if(linepos < 0)
    {
      /* flush buffer to files */
      writelines();
      linepos = 63;
      clearlines();
    }

  outputhex[mybin][linepos]=mychar;

}

void writelines()
{
  int i,j;

  for(i=0;i<NUMOUTPUT;i++)
    {
      fprintf(outputfile,"defparam ram%d.INIT_%02X = 256'h",i,linecnt);
      for(j=0;j<HEXPERLINE;j++)
       {
         putc(outputhex[i][j],outputfile);
       }
      fprintf(outputfile,";\n");
    }
  linecnt++;
}

void clearlines()
{
  int i,j;

  for(i=0;i<NUMOUTPUT;i++)
    {
      for(j=0;j<HEXPERLINE;j++)
       {
         outputhex[i][j]=0x30;
       }
    }
}


/* char1 is the high 4 bits */
unsigned char ascii2bin(int char1, int char2)
{

  unsigned char i;

  i=0;
  if (char1 == EOF)
    return 0;
  if (char2 == EOF)
    return 0;

  if (char1 < 0x41)   /* 0x41 = ASCII A */
    {
      i=(char1 - 0x30);
```

```
    }
  else
    {
      i=(char1 - 0x41 + 10);
    }
  i=i << 4;
  if (char2 < 0x41)   /* 0x41 = ASCII A */
    {
      i=i+(char2 - 0x30);
    }
  else
    {
      i=i+(char2 - 0x41 + 10);
    }

  return i;
}

unsigned char bin2ascii(unsigned char myint)
{
  unsigned char i;

  if(myint > 15)
    printf("%d",myint);
  if (myint < 10)
    {i = myint + 0x30;}
  else
    {i = myint + 0x41 - 10;}
  return i;
}
```

B. LIBXR16.C

```
/*
 * libxr16.c -- minimal xr16 runtime library
 *
 * Copyright (C) 1999, 2000, Gray Research LLC.  All rights reserved.
 * The contents of this file are subject to the XSOC License Agreement;
 * you may not use this file except in compliance with this Agreement.
 * See the LICENSE file.
 */
/* modified 08/25/2008 by David Dwiggins */

typedef unsigned Word;

extern int main();
extern Word  _end;

void _zeromem();
Word* _tos();

/* Processor reset: zero memory and call main.
 */
int _reset() {
/* 08/25/2008 ded, comment out zeromem.  BRAM already in a known state*/
/*     _zeromem(); */

       main();

       /* dynamic halt */
```

```
	for (;;)
		;
}

/* Reset memory to consistent known state.
 */
void _zeromem() {
	Word* p = &_end;
	Word* end = _tos();

	for ( ; p < end; ++p)
		*p = 0;
}

/* Interrupts have not been tested.
 */
int _interrupt() {
	return 0;
}

/* Too clever way to return address of approximately the top of stack;
 * ignore compiler warning on returning address of argument.
 */
Word* _tos(Word arg) {
	return &arg;
}

/* Multiply unsigned times unsigned.
 */
unsigned mulu2(unsigned a, unsigned b) {
	unsigned w = 0;

	for ( ; a; a >>= 1) {
		if (a&1) w += b;
		b <<= 1;
	}
	return w;
}

/* This must be the last symbol in the last module "linked" into the load
 * image.
 */
Word _end;
```

APPENDIX D: ASSEMBLY FILES

This appendix contains the assembly file reset.s used by the linker when generating an XSOC binary file. Reset.s initializes the stack and also contains the interrupt service routine.

A. RESET.S

```
;
; reset.s -- minimal xr16 runtime library startup
;
; Copyright (C) 1999, 2000, Gray Research LLC.  All rights reserved.
; The contents of this file are subject to the XSOC License Agreement;
; you may not use this file except in compliance with this Agreement.
; See the LICENSE file.
;
; This file must be assembled first, so that reset and interrupt receive
; the correct addresses:
; 0000 reset vector
; 0010 interrupt vector; interrupts are untested
;
; tabs=4

global __reset
global __interrupt
global _mulu2

align 16
reset:
       ; perhaps should zero all regs
       ; 08/01/2008 ded
       ; stack pointer set at last word in 8k memory
       ; for CFTP
       lea sp,0x1ffe
       j __reset

align 16
interrupt:
       ; save registers
       sw r1,saveregs+2
       lea r1,saveregs
       sw r2,4(r1)
       sw r3,6(r1)
       sw r4,8(r1)
       sw r5,10(r1)
       sw r6,12(r1)
       sw r7,14(r1)
       sw r8,16(r1)
       sw r9,18(r1)
       sw r10,20(r1)
       sw r11,22(r1)
       sw r12,24(r1)
       sw r13,26(r1)
       ; r14 is the (reserved) interrupt return address
       sw r15,30(r1)
```

```
        ; use interrupted function's stack
        call __interrupt

        ; reload registers
        lea r1,saveregs
        ;lw r2,4(r1)  ; modified 08/31/2008 ded
        ;lw r3,6(r1)  ; add int ack
        lw r4,8(r1)
        lw r5,10(r1)
        lw r6,12(r1)
        lw r7,14(r1)
        lw r8,16(r1)
        lw r9,18(r1)
        lw r10,20(r1)
        lw r11,22(r1)
        lw r12,24(r1)
        lw r13,26(r1)
        ; r14 is the (reserved) interrupt return address
        lw r15,30(r1) ; unnecessary but doesn't hurt to be sure
        lea r3,0xff70 ; interrupt controller at ID 3 +10h
        addi r2,r0,1
        sb r2,0(r3)   ; write a 1 to the ack register
        lw r2,4(r1)
        lw r3,6(r1)
        lw r1,saveregs+2

        ; return from interrupt
        reti

saveregs:
        bss 32
```

APPENDIX E: XSOC LICENSE AGREEMENT

This appendix contains the XSOC License Agreement. The CFTP microcontroller is built upon XSOC; its use requires acceptance of the XSOC License Agreement.

```
XSOC License Agreement Version 1.0
Copyright (C) 1999, 2000, Gray Research LLC.  All rights reserved.

This is a legal agreement between You and Gray Research LLC.
By accepting, receiving, and/or using the Work, as defined below, you
are agreeing to be bound by the terms of this Agreement.  If you do not
agree to the terms of this Agreement, no license in the Work is granted,
and you must destroy, delete or otherwise remove your copy of the Work
and any portion thereof.


INTRODUCTION.

The XSOC Project is a collection of experimental hardware and software
designs and specifications, cited in the Circuit Cellar magazine series,
"Building a RISC System in an FPGA", and providing an example for the
noble purpose of teaching computer design.

DEFINITIONS.

1.  The "Company" is Gray Research LLC.

2.  The "Web Site" is on the World Wide Web at http://www.fpgacpu.org,
    or at such other location as Gray Research may designate from time
    to time.

3.  "You" specifies your person, company, or educational institution.

4.  The "Work" consists of the "Sources" (the documentation,
    specifications, schematics, source code, and scripts that comprise
    the preferred editable representation of the XSOC Project), together
    with the "Outputs" derived from these Sources, including any program
    binaries, programmable logic device configuration data, net lists,
    mask works, and any derivative work based on the foregoing.

5.  An "Excerpt" of the Work is a portion of the Sources, consisting of:
    a) if schematics, up to two original schematic sheets;
    b) if source code, up to fifty lines of original source code.
    c) if neither schematics nor source code, up to two original pages.

TERMS OF GRANT.

The Company grants you a revocable, non-exclusive, royalty-free, and
```

non-transferable license to use, modify, copy, and distribute the Work subject to the following terms and conditions.

1. You shall only use and modify the Work for non-commercial educational and research purposes. You shall not use the Work in any application where failure or error could endanger life or property. You shall not use the Work in a commercial product. You shall not modify the Work to derive a processor design whose instruction set semantics or binary instruction encoding is substantially compatible with any commercially available processor or processor design.

2. You may distribute copies of this entire Work (all files, without modification) provided you do so free of charge.

3. You may distribute an Excerpt or a modification to an Excerpt provided you do so free of charge, you include this license, and you clearly indicate and document the Excerpt as such, and/or your modifications of the Work.

4. With the exception of uses permitted in terms 1-3, you shall not otherwise use, modify, copy, or distribute the Work without the prior written consent of the Company. In particular, you shall not distribute a modified version of the Work via the internet or by other means.

5. If any portion of the Work becomes the subject of a claim of infringement, Company may unilaterally revoke this agreement and any license in the Work granted herein by posting such notice at the Web Site. If the Web Site is or becomes inaccessible, it is your responsibility to use reasonable means to contact the Company in order to determine the status of this Agreement or any revocation of this Agreement.

6. All copies and distributions of the Work must include the above copyright notice and this Agreement in its entirety.

7. This Agreement will terminate immediately upon your failure to comply with any term or condition of this Agreement. Upon termination of this Agrement, you shall make no further use of the Work and you shall destroy all copies of the Work in your possession. The Company shall be entitled to any and all remedies at law and equity, including injunctive relief, for any breach of this Agreement by you. You agree to indemnify and hold the Company, its officers, directors, and affiliates, harmless from all claims, damages, and liabilities resulting from or incident to any use of the work by you.

8. The license herein granted in the Work is and shall remain unilaterally revocable by the Company or its successors in interest in the Work.

9. You agree that any litigation arising under this agreement or in any way relating to this work shall be brought within the State of Washington, and you waive any objection as to venue or jurisdiction of any court located in that State. The governing law of this agreement shall be that of the State of Washington, exclusive of its choice of law principles.

NO WARRANTY.

THE WORK IS PROVIDED UNDER THIS LICENSE ON AN "AS IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED. THE COMPANY SPECIFICALLY DISCLAIMS ANY WARRANTY THAT THE WORK IS FREE OF DEFECTS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE, OR NON-INFRINGING. THE COMPANY DOES NOT WARRANT THAT THIS WORK IS ERROR FREE OR THAT DEFECTS WILL BE CORRECTED. USE AT YOUR OWN SOLE RISK. NO ORAL OR WRITTEN INFORMATION GIVEN BY THE COMPANY OR ITS REPRESENTATIVES SHALL CREATE A WARRANTY INCONSISTENT WITH THIS AGREEMENT. UNDER NO CIRCUMSTANCES SHALL THE COMPANY BE LIABLE TO YOU OR OTHERS FOR DAMAGES OF ANY KIND ARISING FROM ANY USE OF THIS WORK, EVEN IF ADVISED OF POSSIBILITY OF SUCH DAMAGE. NO USE OF THIS WORK IS AUTHORIZED EXCEPT UNDER THIS DISCLAIMER.

CONTACT INFORMATION.

Jan Gray
President, Gray Research LLC
P.O. Box 6156
Bellevue, WA
98008-1156
email: xsoc@fpgacpu.org

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] National Aeronautics and Space Administration, “Single Event Effects,” November 15, 2000, http://radhome.gsfc.nasa.gov/radhome/see.htm, August 2008.

[2] United States Naval Academy, "The Midshipman Space Technology Applications Research (MidSTAR) Program," http://web.ew.usna.edu/~midstar/, August 27, 2008.

[3] Jan Gray, "Building a RISC CPU and System-On-A-Chip in an FPGA," March 7, 2002, http://www.fpgacpu.org/papers/xsoc-series-drafts.pdf, August 25, 2008.

[4] Xilinx Corporation, "QPro Virtex 2.5V Radiation Hardened FPGAs,” DS028 (v1.2), November 5, 2001.

[5] Xilinx Corporation, “Virtex™ 2.5V Field Programmable Gate Arrays,” DS003-1 (v2.5), April 2, 2001.

[6] Dean Ebert, "Design and Development of a Configurable Fault-Tolerant Processor (CFTP) for Space Applications," Master's Thesis, Naval Postgraduate School, Monterey, California, June 2003.

[7] Steven Johnson, "Implementation of a Configurable Fault Tolerant Processor (CFTP)," Master's Thesis, Naval Postgraduate School, Monterey, California, March 2003.

[8] James Coudeyras, "Radiation Testing of the Configurable Fault Tolerant Processor (CFTP) for Space-based Applications," Master's Thesis, Naval Postgraduate School, Monterey, California, December 2005.

[9] Gerald Caldwell, "Implementation of Configurable Fault Tolerant Processor (CFTP) Experiments," Master's Thesis, Naval Postgraduate School, Monterey, California, December 2006.

[10] Xilinx Corporation, “ISE Design Suite Release 9.2.04i,” 2007.

[11] Jan Gray, "The xr16 Specifications," December, 1999, http://www.fpgacpu.org/xsoc/xsoc-beta-093.zip, August 25, 2008.

[12] Chris Zeh, "Incremental Design Reuse with Partitions," Xilinx XAPP918 (v1.0), June 7, 2007.

[13] Carl Carmichael, "Triple Module Redundancy Design Techniques For Virtex FPGAs," Xilinx XAPP197 (v1.0.1), July 6, 2006.

[14] Carl Carmichael, "Correcting Single-Event Upsets Through Virtex Partial Configuration," Xilinx XAPP216 (v1.0), June 1, 2000.

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

[16] Jan Gray, "Getting Started with the XSOC Project," April 6, 2000, http://www.fpgacpu.org/xsoc/xsoc-beta-093.zip, August 25, 2008.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

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

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

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