

Project Number: MQP-BYK-GD07

# Self-Healing Partial Reconfiguration of an FPGA

A Major Qualifying Project Report:

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

---

Evan Custodio

---

Brian Marsland

Date: April 26, 2007

Sponsored by:

General Dynamic C4 Systems

Approved:

---

Professor Brian King

## **Abstract**

The goal of this project, sponsored by General Dynamics, is to create an FPGA-based system capable of detecting and gracefully recovering from errors without compromising system functionality. Previous research developed a prototype for partial reconfiguration, but a major limitation was the need for a PC to partially reprogram the FPGA. By implementing a method of self-reconfiguration and developing a system using triple module redundancy, the FPGA can locate errors and partially self-reconfigure the corrupted areas while maintaining valid system outputs.

## Acknowledgements

We are extremely grateful to both General Dynamics C4 Systems and Worcester Polytechnic Institute for the opportunity to complete this project. Without the cooperation of everyone involved, this project would not have been possible.

We would like to thank Professor Brian King for advising this project in the absence of Professor Berk Sunar. Even though it is not in his usual field of study, Professor King was a tremendous help throughout the term. We are especially appreciative of his efforts to help us complete this project within the constraints of the shortened D-term schedule.

We would also like to thank Professor Berk Sunar for making this opportunity available to us. Although he was unfortunately unable to take part in the specific details of this project, this project and its sponsorship by General Dynamics C4 Systems would not have been possible without him.

We are also grateful for the constant support provided by Brendon Chetwynd and Gerardo Orlando, our on-site advisors at General Dynamics. Brendon provided us with constant support throughout the term, both in technical and administrative aspects of the project. We appreciate Brendon's efforts in allowing us to take the FPGA development board to the WPI Project Presentation Day. We would also like to thank Gerardo for his high-level technical support in this project. He was often a few steps ahead of us in finding potential issues and points of consideration in our design.

Lastly, we would like to thank all of the previous MQP research teams that provided us with crucial background information for the completion of this project. Specifically, we would like to thank Mike Kristan, Brian Loveland, and Rob Sazanowicz, who completed the project "Dynamic Partial Reconfiguration of a Field Programmable Gate Array" in the previous term at General Dynamics. Their project provided a clear, step-by-step flow for achieving their results, which was a crucial starting point for the research and design necessary in our project. We would also like to thank Mike Lundy, who completed the project "A Self-Healing Circuit Implementing TMR" at General Dynamics in the previous year. Although we did not use the specific design implemented in his project, the research that went into his project was greatly beneficial to us.

# Table of Contents

Abstract.....	ii
Acknowledgements.....	iii
Table of Contents.....	iv
Table of Figures.....	vii
1 Introduction.....	1
2 Background Review.....	3
2.1 Field Programmable Gate Array.....	3
2.1.1 The Xilinx Virtex-II Pro XC2VP30.....	5
2.1.2 Single Event Upsets and Single Event Transients.....	6
2.2 Partial Reconfiguration.....	7
2.3 Triple Module Redundancy.....	8
2.4 Practical Uses of Self-Healing Systems.....	13
2.5 Previous Work.....	14
2.6 Tools.....	14
2.6.1 Virtex-II Pro Memec Development Board.....	15
2.6.2 Xilinx EDK 8.2.02.....	15
2.6.3 Xilinx ISE 8.2i.....	16
2.6.4 Xilinx PlanAhead 8.2.7.....	16
3 Project Goals.....	17
3.1 Top-level Goals.....	17
3.2 Technical Goals.....	17
4 Methodology.....	19
4.1 Tool Flow.....	19
4.2 Analysis Strategies.....	20
4.3 Design Strategies.....	20
4.4 Contingency Plans.....	21
5 Implementation of a Self-Reconfiguring System.....	22
5.1 Analysis.....	22
5.1.1 PowerPC or Custom Logic.....	22
5.1.2 Memory Storage for the PowerPC Data and Instructions.....	24
5.1.3 Memory Storage for the Partial Bit Files.....	25
5.1.4 Tool Integration Issues.....	27
5.1.5 Internal Reconfiguration Access Port.....	29
5.2 Design.....	30
5.2.1 Design Intent.....	30
5.2.2 EDK Hardware Considerations.....	31
5.2.3 EDK Software Considerations.....	32
5.2.4 Top-Level.....	34
5.2.5 Bus Macros.....	36
5.2.6 Module Level.....	37
5.2.7 PlanAhead Implementation.....	39
5.3 Production Flow.....	41
5.3.1 Considerations for Further Development in EDK.....	43

5.3.2	Onboard Partial File Storage.....	43
5.4	Testing and Results.....	45
6	Implementation of a TMR-Based Self-Healing System.....	50
6.1	Analysis.....	50
6.1.1	Custom VHDL TMR Implementation vs. TMRTool.....	50
6.1.2	Partial Regions and Safe Regions.....	51
6.1.3	Error Decoding.....	54
6.1.4	Error Sampling.....	55
6.1.5	Output Enables.....	56
6.1.6	PowerPC Partial Module Reset.....	56
6.1.7	DCM Module Placement.....	57
6.1.8	Considerations for Sequential Systems.....	57
6.1.9	Expandability of the TMR Design.....	59
6.2	Design.....	61
6.2.1	Design Intent.....	61
6.2.2	EDK Software Design.....	62
6.2.3	EDK Hardware Design.....	66
6.2.4	Logic Module.....	71
6.2.5	Partial Module.....	72
6.2.6	Safe Module.....	74
6.2.7	Generic Top-Level TMR.....	74
6.2.8	PlanAhead Implementation.....	78
6.3	Production Flow.....	81
6.4	Testing and Results.....	85
7	Future Considerations.....	93
7.1	Bus Macros and Internal Routing.....	93
7.2	PowerPC Wrappers.....	94
7.3	ICAP.....	94
7.4	Software Storage.....	95
7.5	TMR Considerations.....	95
7.6	State Machine Considerations.....	96
8	Conclusion.....	97
	References.....	99
	Appendix A: Self-Reconfiguring System.....	101
A.1:	PowerPC Software.....	101
A.2:	Top-Level – VHDL.....	103
A.3:	Reconfigurable Module – LED Switch – VHDL.....	106
A.4:	Static Module – LED Display – VHDL.....	107
A.5:	Static Module – Clock Converter – VHDL.....	108
A.6:	User Constraints File.....	109
	Appendix B: TMR-Based Self-Healing System.....	114
B.1:	PowerPC Software.....	114
B.2:	PowerPC Wrapper – Top-Level Wrapper – VHDL.....	116
B.3:	PowerPC Wrapper – User Logic – VHDL.....	123
B.4:	PowerPC Wrapper – Clock Converter – VHDL.....	128
B.5:	PowerPC System – VHDL.....	129

B.6: PowerPC System Synthesis Parameters.....	132
B.7: Top-Level – VHDL.....	132
B.8: Reconfigurable Module – Logic and Voters – VHDL.....	138
B.9: Reconfigurable Module – LED Blink – VHDL.....	140
B.10: Reconfigurable Module – Majority Voter – VHDL .....	142
B.11: Reconfigurable Module – Minority Voter – VHDL .....	142
B.12: Static Module – Safe Module – VHDL.....	143
B.13: User Constraints File.....	144
Appendix C: Project Goals .....	147

## Table of Figures

Figure 2-1 – Typical FPGA .....	4
Figure 2-2 – Virtex-II Pro / Virtex-II Pro X FPGA Family Members.....	5
Figure 2-3 – Floor Architecture of Virtex-II Pro .....	5
Figure 2-4 – Memec FF1152 Development Kit.....	6
Figure 2-5 – Truth Table and Schematic of 1-Bit Majority Voter.....	9
Figure 2-6 – Truth Table and Schematic of Minority Voter.....	10
Figure 2-7 – Minority Voters with Tri-state Buffers and Wired-OR gate .....	11
Figure 2-8 – From Outside to Combinational Logic and Majority Voters .....	12
Figure 2-9 – Full TMR Logic Implementation .....	12
Figure 2-10 – Full Self-Healing System Implementation.....	13
Figure 5-1– Interface for the ICAP module.....	29
Figure 5-2 – ICAP Initialization Code.....	32
Figure 5-3 – Main ICAP Programming Function .....	34
Figure 5-4 – XST Process Properties.....	35
Figure 5-5 – Nested Wide Bus Macros.....	37
Figure 5-6 – Matching Port Definitions and Entity Names for a Partial Module.....	38
Figure 5-7 – Bus Macro Placement .....	40
Figure 5-8 – Partial Bit Files Memory Mapping for the ICAP.....	45
Figure 5-9 – Oscilloscope Reading of 2 Hz LED during Switching to Blank Reconfiguration... 46	46
Figure 5-10 – HyperTerminal Output for Blank Reconfiguration.....	47
Figure 5-11 – Oscilloscope Reading of 2 Hz LED during Blank to Switching Reconfiguration. 48	48
Figure 5-12 – HyperTerminal Output for LED-Switching Reconfiguration .....	48
Figure 6-1 – Full TMR Logic Implementation .....	53
Figure 6-2 – Partial Region Boundaries.....	54
Figure 6-3 – Voting on Feedback for Redundant State Machines.....	58
Figure 6-4 – Beginning of main.....	62
Figure 6-5 – The banner procedure.....	63
Figure 6-6 – Infinite loop error checking code .....	65
Figure 6-7 – User-Defined Ports for the Custom PowerPC Wrapper.....	67
Figure 6-8 – Error Decoding Process for the Custom PowerPC Wrapper .....	68
Figure 6-9 – Reset and Enable Mapping for the Custom PowerPC Wrapper .....	69
Figure 6-10 – Configuration Window for the opb_tmr Custom PowerPC Wrapper in EDK.....	70
Figure 6-11 – Feedback Logic for the Blinking LED Module .....	71
Figure 6-12 – Final Voter Output for the Partial Module.....	72
Figure 6-13 – Generic Majority and Minority Voter Instantiation in the Partial Module .....	73
Figure 6-14 – Input Enables for the Partial Module .....	73
Figure 6-15 – Safe Region VHDL Code.....	74
Figure 6-16 – Instantiation of the Input Bus Macros.....	76
Figure 6-17 – Instantiation of the Voter Input Bus Macros for Partial Module 0 .....	76
Figure 6-18 – Instantiation of the Logic and Voter Output Bus Macros for Partial Module 0 ....	77
Figure 6-19 – Instantiation of the Enable Bus Macro for Partial Module 0 .....	78
Figure 6-20 – Concatenation of the Logic and Minority Voter Signals for the PowerPC.....	78
Figure 6-21 – PlanAhead Floorplan for a Self-Healing System .....	79

Figure 6-22 – HyperTerminal Output for Initial Operation with No Errors.....	85
Figure 6-23 – Oscilloscope Reading of Synchronized 1 Hz Logic and Final Voter Signals.....	86
Figure 6-24 – HyperTerminal Output for an Error on Bank A.....	87
Figure 6-25 – Oscilloscope Reading of One Module in Error with Valid Final Output .....	88
Figure 6-26 – Oscilloscope Reading of a Reconfigured Module being Resynchronized.....	88
Figure 6-27 – Oscilloscope Reading of the Outputs of a Module in Error.....	89
Figure 6-28 – Oscilloscope Reading of the Outputs of a Module during Reconfiguration.....	90
Figure 6-29 – HyperTerminal Output for an Error on Bank B.....	90
Figure 6-30 – HyperTerminal Output for an Error on Bank C.....	91



# 1 Introduction

Mission-critical digital systems are often implemented on Field Programmable Gate Arrays (FPGAs), which allow for easy reprogramming of the hardware. FPGAs are especially desirable in these applications due to their cost-effectiveness for relatively small quantity productions and the ease in which they can be updated. The reprogrammable nature of FPGAs presents one of its strongest advantages as well as one of its most significant limitations as compared to an Application Specific Integrated Circuit (ASIC), which is a non-reprogrammable integrated circuit that is customized for a particular use. The upgradeability of an FPGA is especially useful for systems implementing cryptographic algorithms since modifications can be made to the hardware in order to strengthen the module without the need for re-fabricating the entire chip. However, this non-permanent characteristic of FPGAs also causes problems with errors in the system, especially in environment subject to high radiation. Particles may cause portions of the reprogrammable circuitry to change states. FPGAs are therefore more prone to errors in its logic values and in its actual circuitry than a more permanent custom hardware solution such as an ASIC. Thus, in cooperation with General Dynamics C4 Systems, this project implements a proof-of-concept system that is capable of detecting such errors and partially self-reconfiguring these corrupted areas. This “self-healing” system is capable of gracefully recovering from errors while maintaining valid system outputs.

A previous research group at General Dynamics C4 Systems developed a proof-of-concept system implementing partial reconfiguration, which is the capability of reprogramming a portion of an FPGA while the rest of the system remains in operation. A major drawback to their system was that it required an external PC to partially reprogram portions of the FPGA. In order to successfully implement a self-healing system, the first goal of this project is to create a self-reconfigurable system. Such a system would extend the previous term’s research with the capability of an FPGA to reprogram itself without the need for any external PC. Once the implementation of a self-reconfiguring system has been realized, the ultimate goal of this project is to expand the design to include support for detecting and reconfiguring modules in error without affecting the output of the system.

A self-healing system as implemented in this project provides the capability of providing extra security for designated modules on an FPGA. These modules are typically mission-critical

modules that must maintain valid outputs for the successful operation of the system as a whole. For instance, modules implementing encryption or decryption algorithms may utilize these additional security features to deter parties from gaining access to confidential information. Because areas of high radiation increase the likeliness of an error occurring on an FPGA, space systems are another potentially useful application of a self-healing design. While no system can perfectly eliminate the possibility of any errors in an FPGA, the additional levels of security implemented by the self-healing system in this project provide a much lower probability of such an error.

## **2 Background Review**

Throughout the research and development of a system that is capable of detecting and gracefully recovering from errors in its circuitry, many new technologies and concepts have been encountered. Many of these technologies are not yet common practices in digital logic and FPGA (Field Programmable Gate Array) design, and therefore they require significant background research and consideration.

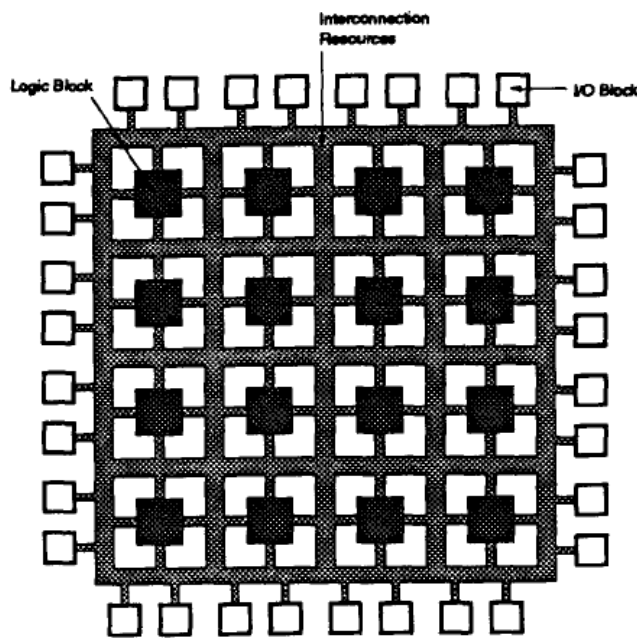
FPGAs are typically used in designs due to their ease of reprogramming; however, this reprogrammable nature causes FPGAs to be prone to errors. As a result of the non-permanent nature of FPGAs, it is possible that portions of the reprogrammable circuit can erroneously change states during operation. This is especially problematic in environments subject to high levels of radiation. Therefore, to achieve a reliable FPGA design, the system must be capable of detecting errors and reprogramming modules as necessary.

Partial reconfiguration is the capability of reprogramming a portion of an FPGA while the rest of the system remains in operation. The concept of partial reconfiguration is a relatively new and uncommon technology used in FPGA design. In order to create a system that has no down-time while repairing modules in error, partial reconfiguration must be used to reprogram these modules without affecting the rest of the system. Triple Module Redundancy (TMR) is used to replicate modules and compare the outputs, which significantly mitigates the probability of that error appearing at the output. TMR can then be used in conjunction with partial reconfiguration to detect errors in a module and reprogram that module as necessary without affecting the final output of the system. The following sections provide a brief overview of these topics and applications of such a system before detailing the design and implementation specifics in later chapters of this report.

### **2.1 Field Programmable Gate Array**

A Field Programmable Gate Array (FPGA) is a user-programmable logic device which can be programmed to interconnect arrays of switches to arrays of logic elements. FPGAs are very useful to developers because of their reprogrammable nature. With an FPGA, companies

have the freedom of modifying hardware designs since the device is reprogrammable. Devices such as Application Specific Integrated Circuits (ASICs) are chips that are built for a certain use and cannot be modified after fabrication. Given the FPGA's reprogrammable nature, it is also easier for companies to issue out firmware updates when the need arises. FPGAs were first created in 1985 by a company called Xilinx. Ever since then FPGAs have grown in popularity and many different companies have been using FPGA technology in all types of products [3]. The main drawbacks of FPGAs compared to ASICs are their high per-unit cost, slower performance and high power consumption [20]. However, as better transistor technology is implemented some of these drawbacks are slowly disappearing.



**Figure 2-1 – Typical FPGA**

Field Programmable Gate Arrays typically work by developing and synthesizing a design on a computer. That design then has all the details on how to route logic gates on the FPGA to allow the creation of the circuit defined in the design. Figure 2-1 illustrates the high-level make up of a typical FPGA.

From Figure 2-1 you can observe that an FPGA contains an entire array of logic blocks. These logic blocks can be as simple as transistors, NAND or XOR gates, or as complex as multiplexers or small microprocessors. The logic blocks are surrounded by programmable interconnections and switches which connect these logic blocks to I/O blocks [14].

## 2.1.1 The Xilinx Virtex-II Pro XC2VP30

The FPGA which is used in this project is the Xilinx Virtex-II Pro XC2VP30. The XC2VP30 has some interesting features to note. Firstly, in comparison to its entire family, the XC2VP30 is a medium-grade FPGA. It has two PowerPC cores, 30,816 logic cells, and 13,696 slices. Figure 2-2 below shows how the XC2VP30 compares to the rest of the members in the Virtex-II Pro family line [16].

Device <sup>(1)</sup>	RocketIO Transceiver Blocks	PowerPC Processor Blocks	Logic Cells <sup>(2)</sup>	CLB (1 = 4 slices = max 128 bits)		18 X 18 Bit Multiplier Blocks	Block SelectRAM+		DCMs	Maximum User I/O Pads
				Slices	Max Distr RAM (Kb)		18 Kb Blocks	Max Block RAM (Kb)		
XC2VP2	4	0	3,168	1,408	44	12	12	216	4	204
XC2VP4	4	1	6,768	3,008	94	28	28	504	4	348
XC2VP7	8	1	11,088	4,928	154	44	44	792	4	396
XC2VP20	8	2	20,880	9,280	290	88	88	1,584	8	564
XC2VPX20	8 <sup>(4)</sup>	1	22,032	9,792	306	88	88	1,584	8	552
<b>XC2VP30</b>	8	2	30,816	13,696	428	136	136	2,448	8	644
XC2VP40	0 <sup>(3)</sup> , 8, or 12	2	43,632	19,392	606	192	192	3,456	8	804
XC2VP50	0 <sup>(3)</sup> or 16	2	53,136	23,616	738	232	232	4,176	8	852
XC2VP70	16 or 20	2	74,448	33,088	1,034	328	328	5,904	8	996
XC2VPX70	20 <sup>(4)</sup>	2	74,448	33,088	1,034	308	308	5,544	8	992
XC2VP100	0 <sup>(3)</sup> or 20	2	99,216	44,096	1,378	444	444	7,992	12	1,164

Figure 2-2 – Virtex-II Pro / Virtex-II Pro X FPGA Family Members

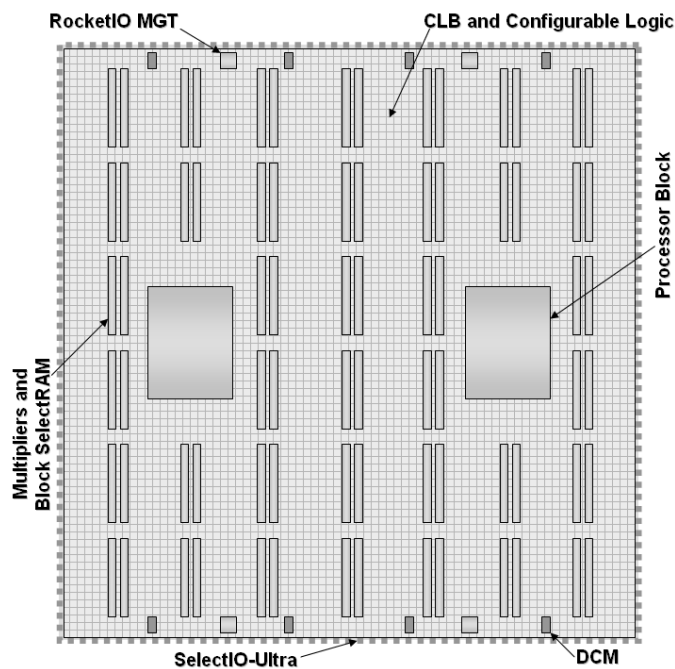
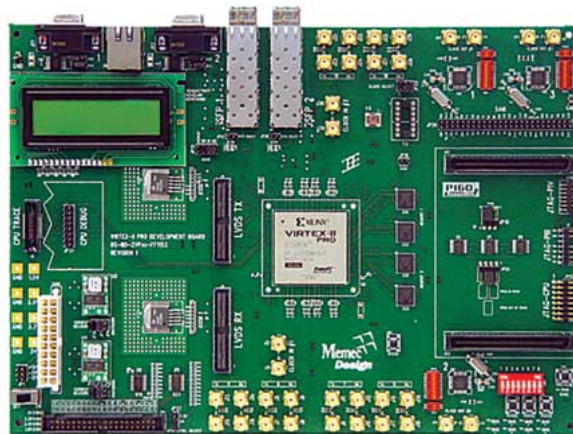


Figure 2-3 – Floor Architecture of Virtex-II Pro

Figure 2-3 above shows an image of the floor architecture for the Virtex-II Pro FPGA. In the figure the two PowerPC cores are visible in the middle of the floor area. There are SelectIO-Ultra and DCM (Digital Clock Manager) interfaces on the side and there is BlockRAM placed vertically along the floor area [21].

Along with the powerful Virtex-II Pro FPGA this project will use the FPGA on a Memec development board. This development board has plenty of I/O to use. Aside from simple LEDs and buttons, this development board contains 32MB of SDRAM, 2 x 16 character LCD, 10/100 Ethernet PHY, JTAG, CPU debug, CPU JTAG ports, RS-232 and more. Figure 2-4 below is an image of the development board used in this project [1].



**Figure 2-4 – Memec FF1152 Development Kit**

### **2.1.2 Single Event Upsets and Single Event Transients**

One of the major motivations of this project is to combat system critical errors to FPGA designs. There are two main types of errors that are being considered, the Single Event Upset and the Single Event Transient. The Single Event Upset occurs when radiation affects the transistors that are part of the look up table logic of the FPGA. If the lookup table becomes affected by radiation it can change bit values associated with the hardware makeup of the current FPGA design. If radiation does modify a lookup table, this could cause a NOR to be switched to a NAND therefore compromising the entire system [5].

The Single Event Transient is less damaging than the Single Event Upset mainly because it does not affect the hardware makeup of the current FPGA design. The Single Event Transient instead affects current processing of data in the circuit. Radiation can hit areas on the FPGA and cause an incorrect bit value to the input or output of critical modules. Although this does not affect the hardware makeup of the FPGA design, many mission critical designs require high reliability of data processing and in turn these types of errors could just be as dangerous [4].

## 2.2 Partial Reconfiguration

Partial reconfiguration (PR) is the ability for a portion of an FPGA to be reprogrammed while the remainder of the system remains unchanged. A partial bitstream loads only a portion of the design onto the FPGA rather than rewriting the entire design. Partial reconfiguration is especially useful for reprogramming a portion an FPGA during operation without affecting the rest of the system. This practice is called dynamic partial reconfiguration. Static partial reconfiguration refers to reprogramming a portion of the FPGA while the rest of the board is in a reset state [21]. Dynamic reconfiguration is much more useful since it allows device reconfiguration during runtime. Therefore, any future use of the term “partial reconfiguration” in this paper will refer to dynamic partial reconfiguration.

Xilinx supports partial reconfiguration for all of its FPGAs, ranging from the Virtex-4 devices to the low-cost Spartan-3 boards. The two main methods of partial reconfiguration supported by Xilinx are module-based and difference-based. Module-based partial reconfiguration is used to specify portions of the device that can be reconfigured, which are known as reconfigurable modules. This module-based design typically requires floorplanning to specify where all of the reconfigurable modules will be placed on the physical layout of the FPGA. When the device is to be reprogrammed, the entire reconfigurable module is overwritten with the new partial bitstream. Difference-based partial reconfiguration is used to only make small changes in the FPGA design. The generated bitstream only includes differences between designs. Difference-based partial reconfiguration allows for faster reprogramming of the device since only the changes must be rewritten, but it has limited applications as compared to the module-based solution. In this project, only module-based partial reconfiguration will be used [9].

In a system implementing module-based partial reconfiguration, modules that are to be kept in continuous operation without the capability of being partially reprogrammed are referred to as static modules. One or more modules can be designated as the partially reconfigurable module(s), which require additional considerations in the design, synthesis, and implementation stages. Specifically, a modular design flow must be used which will synthesize and create separate bitstreams for each of the reconfigurable modules as well as a total system bitstream including all of the static logic and one implementation of each of the reconfigurable modules.

Partial reconfiguration can be implemented through a JTAG connection to a PC or internally through custom logic or an on-board processor, such as the embedded PowerPC in the Virtex II Pro FPGA. Partially reprogramming the FPGA through internal circuitry, referred to as self-reconfiguration, is a much more useful method of partial reconfiguration since it eliminates the need for an external PC. The partial bitstreams are stored in memory and are written to the Internal Configuration Access Port (ICAP) of the FPGA in order to reconfigure the specified region of the board with the new logic. Self-reconfigurable modules allow for the implementation of a self-healing device, which is a system capable of detecting errors in its own circuitry and reconfiguring the modules as necessary. A self-healing device can detect and gracefully recover from errors while maintaining valid output from the system. This is especially valuable for devices that are operating in the field and contain crucial applications such as circuitry implementing cryptographic algorithms, where errors in such modules could cause access to information that is supposed to be kept confidential.

## **2.3 Triple Module Redundancy**

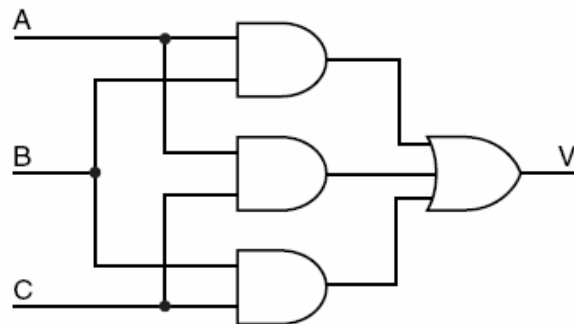
FPGAs allow a hardware engineer tremendous flexibility to implement and revise hardware designs in a very short amount of time. The FPGA's soft-core nature makes it possible to route arrays of logic elements in a certain fashion according to a bit map software description. Unfortunately, as is the problem with many large memory arrays, the FPGA can be affected by external radiation. This radiation can cause corruption with the interconnections between logic elements. This issue has always been a major concern in unknown high-radiation applications (such as space). Comparatively, this radiation is a lesser concern in ASIC designs because most of the connections between logic elements are physically hardwired together, and not connected



by using programmable switches. The radiation is dangerous to the FPGA because the state of these programmable switches in charge of routing can flip as any outside radiation hits it [4]. For instance, if an FPGA was programmed to route two inputs into an OR gate, radiation could hit the programmable switch in charge of this routing and the inputs routed into the OR gate may then be routed into an AND gate. These types of events can be classified as an SEU. Triple Module Redundancy with a majority vote is an ideal method to prevent these SEUs from affecting the FPGA's programming.

The concept of Triple Module Redundancy (TMR) is based on the idea of implementing the same logic task three times. The three outputs from these redundant modules are then compared in order to locate any possible area corruptions. Ideally the three outputs should be all identical, however if this is not the case then these three outputs are compared using the majority voter. The majority voter takes an input from three individual but redundant modules in parallel. If two or more of these inputs are equal, then the output of the majority voter is input majority. Figure 2-5 below is a schematic and truth table of a 1-bit majority voting circuit. This majority output is decided by the majority voter to be the true output of this redundant logic. The major benefit from this is that if one out of the three modules is not in working condition, the overall functionality of the logic is not compromised.

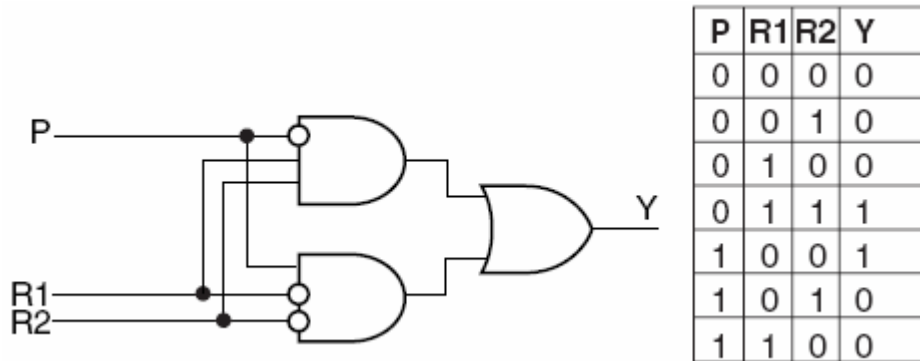
A	B	C	V
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



**Figure 2-5 – Truth Table and Schematic of 1-Bit Majority Voter**

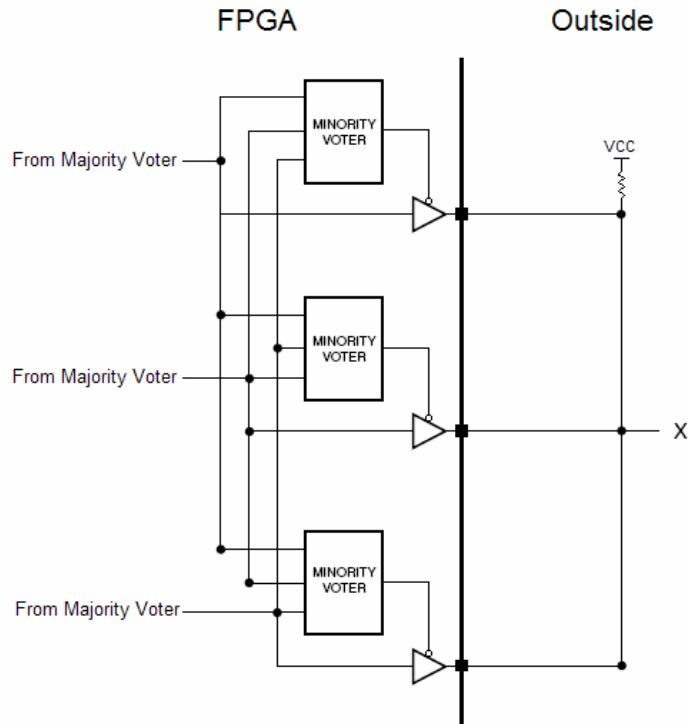
This method of using TMR with only one majority voter circuit is still flawed, this is because the SEU not only could affect the redundant modules but can also affect the voting circuit itself. To alleviate this issue the majority voting circuit must also be redundant. These redundant majority

voters must be compared using minority voter circuits. The minority voters also take in three inputs, the primary path and two other redundant paths in question. If the primary path is in the majority with one other redundant path then the output is low. If the primary path is in the minority in comparison with the two other redundant paths then the output is high. Figure 2-6 below is a schematic and truth table of the minority voting circuit.



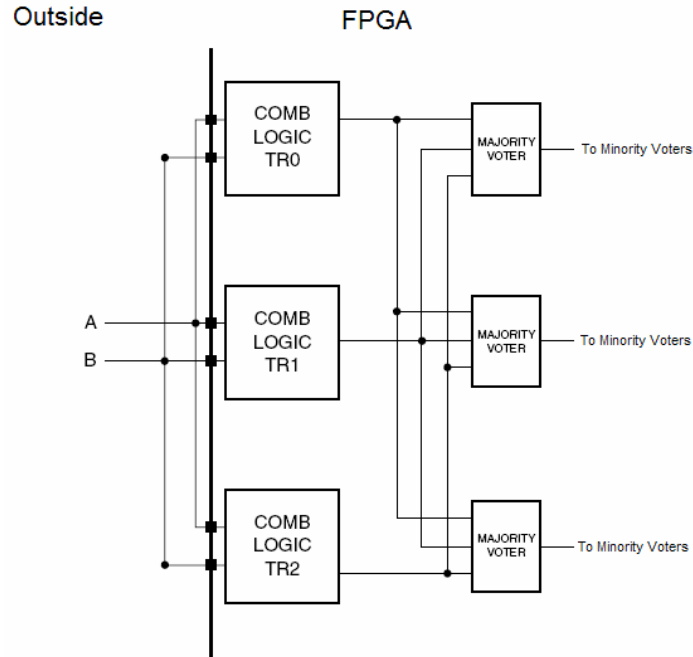
**Figure 2-6 – Truth Table and Schematic of Minority Voter**

This minority voter output is fed into the control signal of a tri-state buffer with an inverted control input. If the path in question is the minority then the tri-state buffer will be placed into high-impedance. If the path in question is in the majority then its corresponding tri-state buffer will allow the path to follow through to output. These three outputs will connect together outside of the FPGA into a wired-OR fashion. Figure 2-7 shows the minority voters controlling the tri-state buffers which feed outside to the wired-OR gate.

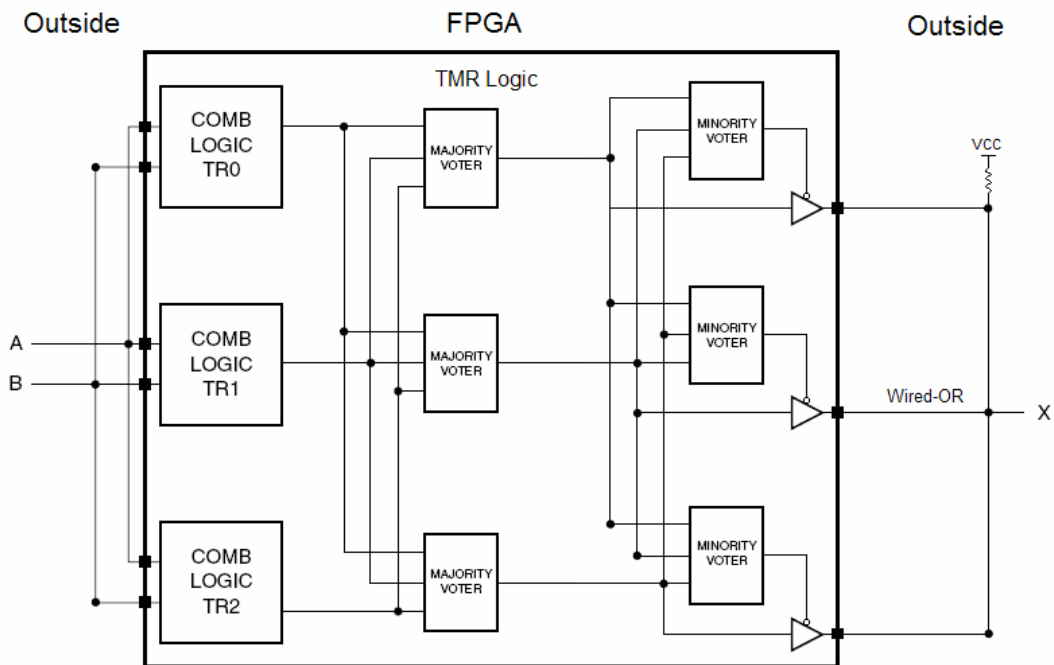


**Figure 2-7 – Minority Voters with Tri-state Buffers and Wired-OR gate**

The main reason of implementing the Wired-OR on the outside of the FPGA is because it is the main circuit which combines all TMR outputs. It is the one section which must be hardwired so that it cannot be affected by the SEUs. If the three outputs were connected inside the FPGA, this would still be considered a soft-core wire connection and can be compromised by SEUs. Similarly, the inputs must be duplicated outside of the FPGA in order to guarantee true signal independence. Figure 2-8 below shows the hardwired inputs into the FPGA TMR logic.



**Figure 2-8 – From Outside to Combinational Logic and Majority Voters**



**Figure 2-9 – Full TMR Logic Implementation**

This full TMR implementation can be shown in Figure 2-9. The full TMR design example illustrates how to implement TMR with only combinational logic. Special

considerations must be taken into account when implementing TMR with simple and complex state machines [4].

## 2.4 Practical Uses of Self-Healing Systems

Partial Reconfiguration and Triple Module Redundancy can be used together to create a true Self-Healing system design. In order to do this, the full TMR implementation must be extended to provide additional status outputs to the partial reconfiguration controller. During FPGA operation as data is being processed through the redundant modules, if any of the three modules gets corrupted the overall output is not compromised, also the status signal will be set to communicate to the partial reconfiguration controller in order to configure the module in question. During this time there is no downtime or effect on the overall system while a module is being reconfigured. Figure 2-10 below illustrates the full system diagram of a Self-Healing system.

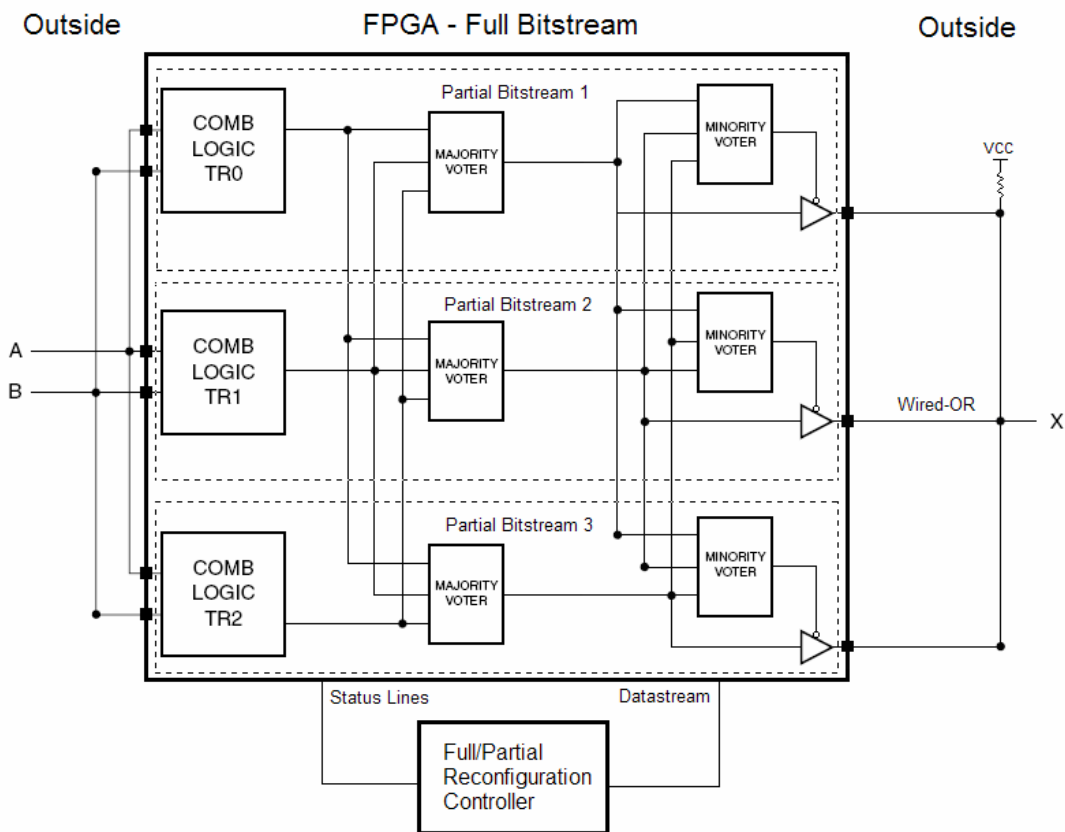


Figure 2-10 – Full Self-Healing System Implementation

A practical use of this system design would be in cryptography. Implementing a cryptographic algorithm on an FPGA provides exposure to a couple of known attacks in order to break the integrity of the algorithm. Such attacks are referred to as side-channel attacks. One may be capable of corrupting portions of the FPGA in order to analyze its behavior. Analyzing such behavior can give an attacker insight on how to compromise the security. Implementation a Self-Healing system to the cryptographic algorithm can prevent corruption to the FPGA and alleviate the possibilities for this type of attack [12].

Another practical use of this system design would be in any type of high-radiation setting such as space. There are many radiation particles traveling through space, if no protection is applied, these particles can strike your device and possibly cause it to malfunction. A Self-Healing system using Partial Reconfiguration and Triple Module Redundancy will drastically reduce the probability of radiation causing complete system failure and will ensure full system uptime during the healing process.

## **2.5 Previous Work**

Previously completed Major Qualifying Projects at General Dynamics C4 Systems have provided a framework for the research and design to be completed in this project. The project, “Dynamic Partial Reconfiguration of a Field Programmable Gate Array” completed in March 2007, detailed the design flow for a system implementing partial reconfiguration as well as an initial study of potential implementations of self-reconfiguration. “A Self-Healing Circuit Implementing TMR,” completed in April of 2006, provided a background on the use of TMR for error detection and correction in FPGAs. This project ultimately combines these two background areas of research into a complete self-healing cryptographic system.

## **2.6 Tools**

Several hardware and software tools are necessary for the completion of this project. These tools include the physical FPGA and its associated development board that allowed for

continual reprogramming of test systems as well as many features for data storage and output display. Xilinx has also supplied a suite of tools that are necessary for this project. These Xilinx software tools are used for developing the hardware and software aspects of the system. Although many of these tools have included documentation from Xilinx, their support of partially reconfigurable systems is currently somewhat lacking. Therefore, the integration of these tools into a working tool flow to achieve the goal of a self-healing system required research from numerous sources and some experimentation with the tools.

### **2.6.1 Virtex-II Pro Memec Development Board**

The Virtex-II Pro Memec Development board provides a variety of features in order to implement our partial reconfiguration designs. This board contains a Virtex-II Pro FPGA chip from Xilinx which has two PowerPC cores. Out of the many features which this board provides, this project just utilizes the FPGA, a PowerPC core, onboard memory storage and the inputs/outputs provided by RS-232, LEDs, and dip-switches.

### **2.6.2 Xilinx EDK 8.2.02**

Xilinx EDK is a software development kit used to utilize the functionality of the onboard PowerPC cores of the Virtex-II Pro FPGA chip. The EDK provides us with IP Cores to bridge functionality of hardware designs on the FPGA with software designs on the PowerPC. This is an attractive feature as this project will implement VHDL designs on the FPGA while being able to partially reprogram those designs using the PowerPC. Most of the software developed for this project will be created using the Xilinx EDK interface. After a successful software build, the EDK can synthesize the correct VHDL wrappers to instantiate the PowerPC core and can inject the PowerPC instructions into a blockram on the FPGA for the PowerPC to read and process.

### **2.6.3 Xilinx ISE 8.2i**

Xilinx ISE is a popular FPGA development tool used widely in industry and in educational institutions. This tool allows for complete FPGA development. ISE can read in VHDL/Verilog/Schematic modules created for a project design and synthesize them into logic elements to be placed on an FPGA. ISE can automatically interpret your HDL syntax, synthesize your description, place and route the logic elements and then provide a software BIT file description to connect these logic elements together to create the circuit described in the HDL. All these tool flow steps require their own respective application program to perform the function. ISE calls these programs automatically in a pleasant GUI so that the user can create FPGA designs in seconds.

### **2.6.4 Xilinx PlanAhead 8.2.7**

Xilinx PlanAhead is a floorplanning tool provided by Xilinx to allow developers flexibility on how their synthesis designs should be placed on the FPGA floorplan. This tool is useful in ASIC designs where locations of logic elements are an important factor to the performance of the application. For the scope of this project, PlanAhead has partial reconfiguration options which make it a required tool in the partial reconfiguration tool flow.



## **3 Project Goals**

Before the start of the project certain top-level and technical goals were laid out. The top-level goals must satisfy both General Dynamics C4 Systems and WPI. The work done must provide General Dynamics with enough research and documentation so that the project could be re-created if the need arises. Also, the work must provide the project group enough educational benefit to merit a grade as a Major Qualifying Project for WPI. Once the top-level goals were set, the technical goals were decided. These goals are very specific to the type of research that needed to be done.

### **3.1 Top-level Goals**

The project group wanted to give General Dynamics C4 Systems a fully implementable self-healing design using partial reconfiguration and triple module redundancy. It was also decided that this design should have the ability to be re-created with helpful step-by-step documentation. The project group expected that General Dynamics would use this design for future self-healing crypto applications in high radiation environments. In order to satisfy WPI's goals the project group would need to gain lots of experience with VHDL, Xilinx's tool environment and development on the PowerPC core in order to research and write a comprehensive academic report.

### **3.2 Technical Goals**

Our main overall technical goal was to implement a cryptographic circuit which includes triple module redundancy for error checking and has the capability of healing itself if an error were to be detected. This main top-level technical goal is split into three important milestones. First goal is to design a self-reconfiguring system, the second goal after is to design a self-healing system and the last goal is to design a more complex VHDL module for the self-healing system. Each of these goals and their sub-goals will be discussed in more detail.

Designing a self-reconfiguring system required specific technical goals to be met. First was to re-implement the previous project group's design of partial reconfiguration. Next was to become more familiar with the PowerPC and Xilinx Platform Studio. After that, achieving communication with the ICAP port must be obtained. This can either be done using the ICAP IP core of the PowerPC or designing a custom VHDL solution which can interface with the port. After communication with the ICAP port is a success, the project group must implement a simple ICAP design into a partially reconfigurable environment. Once these components are integrated, the next goal would be to load a partial bitstream from flash memory through the ICAP port to the partial area of the FPGA on a button push.

Once the self-reconfiguring system has been design, the next goal would be to design a self-healing system. To do this the project group must implement TMR functionality to detect errors in the redundant modules. This can be done one of two ways, either by using custom VHDL to replicate modules and create voting circuitry or use the Xilinx XTMR tool to create the TMR system. In either case, once this goal is completed the next goal is to modify the ICAP design to load a partial bitstream to a redundant module when an error in the module is detected.

Once the self-healing system has been designed and tested, the next goal is to design a more complex VHDL module to be used in this self-healing system. This goal can be achieved by either using a supplied AES implementation that can fit within the size constraints or by creating an alternate custom algorithm. After all these goals have been met, the project is completed and ready for testing and documentation. For a list of all the technical goals and their estimated difficulty, please refer to Appendix C.

## **4 Methodology**

In this section the methodology that the project group used in completing this project will be discussed. It is important to note that before starting a project, members of the group must have an overall plan on how to engage problems at hand. Not having a pre-defined plan would have cost the group many wasted hours of uncompleted work. This project group could not afford losing any time because of the nature a seven-week Major Qualifying Project. In the following sections the importance of tool flow, analysis strategies, design strategies and contingency plans will be discussed.

### **4.1 Tool Flow**

Tool flow is a very important process to be documented. One of the first goals which this project team had to accomplish was to re-create the partial reconfiguration implementation which the previous project group had achieved. This goal was a complete success because of the tool flow which the previous project group had supplied. It is important to note that this project is not solely research; it is also a step-by-step guide on how to re-create the proof-of-concept of the research. Such a step-by-step guide could not have been possible without a well-documented tool flow.

For each of the goals accomplished, the project group made sure that at the end of the day each step of the successful tool flow was well-documented. This was done for a couple of reasons, since the project group is dealing with complex designs, these tool flows become very lengthy. If the project group did not document the tool flow as goals were accomplished, it would have been difficult to recall all the small mundane steps taken to achieve the goal. The second reason is that if the project group did not document tool flow, and could not recall certain steps in the tool flow, then it would have cost the group valuable time and effort in re-creating designs.

Two tool flows have been created for this project, one for a self-reconfiguring system and one for a self-healing system. The project group hopes that this tool flow will be used by General Dynamics and any other MQP group continuing research in this area.

## **4.2 Analysis Strategies**

When analyzing problems to achieve a goal, the project team must extensively research these problems. To do this, the project team took use of parallel research. Since the project group had only two members, the resources of both members had to be utilized to their full extent. In a given day, if the group had three different things to analyze, one group member would research one of the topics while the other group member would research a different topic. In the end a group meeting would take place to discuss the findings and consider them for implementation of the design. Some topics of analysis would require much more extensive research. For these topics, both group members would research the topic in parallel in order to not follow the same train of thought and to arrive at the solution faster than usual.

## **4.3 Design Strategies**

After analysis was completed on a certain project goal, the project group would start designing. Given that most of the design work was done in either VHDL or C, the group took use of the software engineering concept of pair programming. Different from parallel research, pair programming utilizes the resources of both project members at the same time on one design topic. In pair programming, one of the project members is the main programmer and the second project member reviews all the code written by the first member while it is being written. This strategy was a very efficient use of resources. It combined the intelligence of both project members during the design stage in order to quickly build the designs needed. If the main programmer was stuck on a problem at hand, both members would suggest and build upon each others' ideas to come to a solution. Also, pair programming reduced time on debugging, while the main programmer was writing code, the other project member could review the code as its being written and point out any possible errors that were made.

## 4.4 Contingency Plans

In order to prevent complete dead-lock of the project, the project group made sure that contingency plans were set in place. Since this project required a significant amount of research in order to discover tool flow, there was no guarantee that at any point that a certain plan of action would work. If the project group were to explore a path of research that did not lead to a solution in a reasonable amount of time, then the project would be placed in complete dead-lock. Pre-defined contingency plans solved the problem of project failure. It is important that the project is not placed on a completely sequential goal-path. The project group made sure that for certain goals which may lead to problems, that there are two different paths to come to the same solution. This way if one path is not leading to the solution, the project group can switch paths seamlessly with very little time wasted. Fortunately, this project group did not face any critical research dead-lock with any of the technical goals placed. Good analysis and design strategies used by the project team ensured that all goals were met in order to complete the project.

## **5 Implementation of a Self-Reconfiguring System**

A system implementing dynamic partial reconfiguration over the JTAG connection to a PC was developed by the previous project group. However, the use of partial reconfiguration is not truly valuable unless the FPGA is capable of reprogramming itself independently of an external PC and ultimately without any necessary human interaction. A design that implements self-reconfiguration is a significant milestone for achieving a self-healing system that can detect errors in its circuitry and reprogram the modules as necessary to fix these errors.

### **5.1 Analysis**

The implementation of a self-reconfiguring system requires careful consideration of several significant design choices. The study of self-reconfiguring systems is still in its infancy, so much of the documentation is fragmented, incomplete, or incorrect. Systems implementing partial self-reconfiguration are still uncommon in most FPGA applications due to the relatively large overhead required for the initial analysis phase to find a working tool flow. Xilinx has recently been attempting to make partially reconfigurable systems easier to implement. Xilinx has added more documentation to their website and has produced a more user-friendly floorplanning tool used for partial reconfiguration, but much of this information is currently limited to an “Early Access” area that requires approval for access. Although a more standardized and user-friendly tool flow for partial self-reconfiguration may be available soon, a large amount of research and trial-and-error had to be used in order to attain a self-reconfigurable system using the currently available tools and documentation.

#### **5.1.1 PowerPC or Custom Logic**

A self-reconfiguring system on a Xilinx Virtex FPGA is implemented by making use of the Internal Configuration Access Port (ICAP). A partial bitstream is written to the ICAP, which then reconfigures the specified portions of the FPGA with the new logic. Communication with

the ICAP can be implemented through either the embedded PowerPC or through a custom VHDL logic design.

The Virtex-II Pro FPGAs used in this project include two embedded PowerPCs. One significant advantage to using the PowerPC is that Xilinx has provided an IP core that interfaces to the ICAP. Using this interface, the PowerPC can communicate with the ICAP through simple C method calls, with most of the low-level interface design being hidden to the user. In fact, once the bitstream is loaded into memory, only one single method must be called to successfully reprogram a reconfigurable module on the device. Conversely, a custom logic design would require a manual implementation of the low-level interface to the ICAP, including any necessary timing considerations. Also, the PowerPC provides additional IP cores that enable easy interfacing with the different types of memory in the system as well as the I/O modules of the test board. The use of any of these interfaces would require VHDL code if the PowerPC were not being used. The RS232 port can act as the standard input and output of the C programs, which allows for status messages to be sent over the RS232 cable and displayed on HyperTerminal running on the Windows XP PC. The ability to print status messages during the debugging stage of the design was very useful to quickly determine exactly where an error occurred.

Although the IP cores provided by Xilinx are very useful to hide the low-level implementation of the peripherals, they do make use of more board resources than may be necessary for this application. The PowerPC itself is a hardware module that cannot be reconfigured on the FPGA, but the wrapper files associated with the IP cores utilize many additional board resources. A custom logic interface would be less resource intensive since only the required logic would be included in the design. The IP cores include functionality for some peripherals that are never used in this project. Also, once the system is implemented with TMR in the next portion of the project, the PowerPC introduces some additional issues. The PowerPC itself does not have to be replicated since it is actually part of the hardware and not part of the reconfigurable logic; however, the surrounding VHDL wrapper files are susceptible to errors caused by SEUs. It is difficult to include these wrapper files in the replicated portion of the design, so they are not protected from SEUs.

The PowerPC was chosen as the method for interfacing with the ICAP in this design. Although it does use more resources than necessary, the custom VHDL modules implemented in

this system are relatively small and are not limited by board resources. Additionally, the time it saved by hiding the low-level details of the ICAP and other peripherals allowed for other areas of the project to be reached that may not have been otherwise possible.

### **5.1.2 Memory Storage for the PowerPC Data and Instructions**

When using the Xilinx EDK tool to implement PowerPC functionality on the Virtex-II Pro FPGA, the Block RAMs internal to the FPGA are automatically initialized with the proper program data and instruction memory when the bitstream is generated and written to the board. The block memory contents are specified in a Block Memory Map (BMM) file, which contains a syntactic description of how individual Block RAMs (BRAMs) create a contiguous logical data space. During the Place and Route (PAR) and bitstream generation portions of the design implementation, the BMM template file is updated to specify the configuration of the BRAMs. However, when utilizing the self-reconfiguration design flow, specifically in the PlanAhead software, this process does not initialize the BMM files correctly. The solution to this problem requires updating the BMM template file automatically generated by EDK with the updated design hierarchy as well as modifying the scripts generated by PlanAhead to include the BMM file. The exact changes will be specified in the Production Flow of the project report. Once the updated BMM file has been generated, it must be imported back into the EDK project directory along with the new full bitstream. EDK will properly read the updated bitstream and BMM files, and it will automatically initialize the Block RAM modules when the bitstream is uploaded to the FPGA. This allows for easy software reprogramming since the data and memory initialization will be automatically loaded

An alternative method for storing program data and instruction memory is to use the Static RAM (SRAM) modules on the Memec test board external to the FPGA. These SRAM modules are much larger than the Block RAM internal to the FPGA. Although this design does not use a significant portion of data or instruction memory, future designs may be limited by the size of the Block RAM. Also, since the Block RAM modules are part of the FPGA, they are susceptible to SEUs, which may be problem in a TMR design. Since the SRAM modules are external to the FPGA, they are not as vulnerable to these problems. However, because they are not part of the FPGA, they cannot be programmed directly through the JTAG interface. Instead,



data must be written to the FPGA through the JTAG interface and then transferred to the external SRAM modules. This can be done through the “Program Flash Memory” menu option in EDK. Unfortunately, several errors were encountered when attempting to use this method, and the results are inconsistent between different EDK projects. Each time a program is loaded to the board, the SRAM must be configured separately, unlike the Block RAM modules which are configured automatically in the bitstream.

The self-reconfiguring system makes use of the Block RAM modules internal to the FPGA in this design. Although the Block RAM modules are smaller and more error-prone than the SRAM modules, they are easier to use in EDK once the design has been implemented and the final bitstream and BMM files have been created.

### **5.1.3 Memory Storage for the Partial Bit Files**

Similar to the previous design consideration concerning where program code should be stored in memory, a separate decision must be made concerning the storage of the partial bit files in the self-reconfigurable system. Partial bit files are typically equal in size or larger than program code in the simple applications implemented in this project. The bit files for relatively simple modules have generally ranged from 50KB to 75KB in size, and larger, more complicated modules will have larger partial bit file sizes. The program code is typically around 60KB in size, but will likely not significantly increase in size as much as the partial bit files. The code for reprogramming the ICAP should remain relatively consistent regardless of the size of the bit files. Also, when configuring the system using the start-up wizard in EDK, 128KB of Block RAM can be reserved for program code while only 64KB can be reserved for program data. Since multiple partial bit files will be stored in memory consecutively, a larger memory space is necessary for storing the bitstreams than was necessary for storing the application code.

The 4MB Flash Memory module was chosen to store the partial bit files. This memory module is external to the FPGA and is part of the Memec test board. Unfortunately, the use of the external Flash Memory did complicate the tool flow. Some projects created in EDK were unable to properly write data to the Flash Memory while other projects did not cause any problems. Another issue that was encountered was that partial bit files are binary files, which would often cause the flash memory programming tool to freeze without any indication of an

error. In these cases, though, an ASCII file could be programmed to the Flash Memory; however, when reading the file back, only the first 16KB of the file was programmed due to the 16KB sector size of the Flash Memory module. An ASCII file would have to be reverted to binary data in the Flash Memory, so this solution was not desirable even if the sector size problem could be alleviated.

The initial solution to this problem involved using the WinHex software hex editor to convert the binary files into a C array [17]. This array was copied into the program code as a global variable so it would be stored in a Block RAM module. The PowerPC would then use a simple loop to program each byte in the array to consecutive memory locations. Since the Flash Memory is non-volatile, this process of creating C-style arrays and programming them to memory only had to occur once for each partial bit file until a new design was to be loaded to the board. This method of writing the partial bit files to memory allowed for continued development of the self-reconfigurable system, specifically for communication with the ICAP. An initial complete self-reconfigurable system was implemented using this bit file storage method. Since the array must be temporarily stored in Block RAM, though, it is subject to the same size constraints as previously discussed. This method does not scale well to larger bit files, since those may have to be divided into smaller file, and is therefore not an ideal method of programming partial bit files to Flash Memory.

A complete sample EDK project was provided online for the specific Memec FF1152-6 Virtex-II Pro Rev. 3 Development Board with the additional P160 Communications Module 2 [1]. This project is supplied by the vendor for the exact revision of the board in use for this project. Once this project was loaded into EDK, the “Program Flash Memory” tool in EDK allowed for successful programming of binary partial bit files to the memory. This also alleviated the problem of the 16KB sector size that was previously encountered. It has not yet been determined what features of this supplied project allow for the successful programming of Flash Memory that cannot be completed in other EDK projects. It is possible that the address space set in previous projects was incorrect, or an additional module or configuration file is provided in this project. Determining the exact cause of this error is not a forefront issue in the scope of the MQP, so this EDK project has been used to program the non-volatile Flash Memory with the partial bit files while other EDK projects have been used to include custom hardware and software necessary for self-reconfiguration.

One final issue faced while writing the partial bit files to the Flash Memory is that when writing a file to memory at a specified offset, the entire memory module is cleared before writing the new file. Therefore, multiple bit files cannot simply be programmed separately at different offsets. Instead, the three files must be concatenated into a single file using a hex editor prior to programming. Then, this complete file can be programmed to the Flash Memory.

#### **5.1.4 Tool Integration Issues**

The previous term's Major Qualifying Project, "Dynamic Partial Reconfiguration of a Field Programmable Gate Array," detailed a tool flow to achieve partial reconfiguration using the JTAG interface. The partial reconfiguration tool flow primarily makes use of Xilinx ISE for designing and synthesizing VHDL modules as well as the PlanAhead software for placing and routing the partially reconfigurable design. The implementation of a self-reconfigurable system using a PowerPC requires the additional Xilinx Platform Studio (XPS) and Embedded Development Kit (EDK) tool to be included in the tool flow. The prior term's team also performed some initial research and testing to include EDK into the tool flow to create a self-reconfigurable system, but they were unable to communicate with the PowerPC in their design. A Xilinx document on the "Early Access" partial reconfiguration website details the integration of EDK into the tool flow [7], which is the document that the previous term's group followed. This method involves modifying VHDL files created by EDK to create the necessary top-level module, which includes the static PowerPC module and any other static or reconfigurable modules. Last term's group encountered several problems when attempting to implement this tool flow. One major problem was that EDK would often overwrite files that they modified, specifically the top-level VHDL file and the synthesis parameters file [10]. They also were unable to make the processor boot or gain access to any of the peripherals. They hypothesized that the PlanAhead software broke the I/O connections to the microprocessor hardware [10]. Due to these issues, an alternative tool flow approach was explored in this project.

Several possible tool flows were explored before finding one that was successful. In order to simplify the process, an incremental design approach was used where each step only added a limited amount of additional functionality to the design. First, a simple design using only EDK was used. Next, this design was then incorporated into a design with other static

VHDL modules. A system was then designed that included both the PowerPC along with reconfigurable modules, but the PowerPC did not control the reconfiguration. Finally, the self-reconfigurable circuit would be implemented where the PowerPC controls the reconfiguration of the modules. This incremental design approach allowed for easier debugging since it was easier to isolate exactly what caused an error rather than attempting to create the complete self-reconfigurable system in one step.

Once a project was built using only EDK, the next step was to incorporate this project with other static VHDL modules, without any communication between these modules and the PowerPC. Initially, netlist files generated by EDK were imported into PlanAhead along with several ISE netlists. However, since there was no top-level file incorporating both modules, this design did not work. Next, the research team attempted to export the project from EDK to Xilinx ISE. This would create a folder of the necessary VHDL, synthesis, BMM, and other related files necessary for ISE to synthesize the design. Although this method may have worked, Xilinx has deprecated this method and prefers that the .xmp EDK project file be added to the ISE project instead. This is a much better method since changes in the EDK project will automatically be changed in the ISE project rather than having to export and import a new folder of files. Once the XPS project file has been added to the ISE project, a top-level VHDL file must be created that incorporates any static logic modules with the XPS system component. The ports for the XPS system component are found in the “system\_stub.vhd” file located in the “hdl” folder of the xps project. Component instantiation and port-mapping are performed as with any other VHDL component, and the necessary top-level I/O and user constraints file are likewise modified. Once this has been completed, the complete system is synthesized in ISE as normal, and the resulting netlists are imported into PlanAhead for placing and routing. The remainder of the flow is unchanged, and the system works as expected.

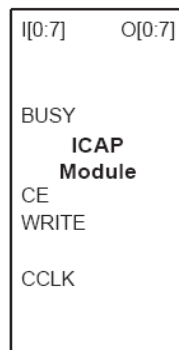
The next step of the incremental design process was to incorporate the reconfigurable modules, although they would not be reprogrammed by the PowerPC. Although the synthesis and implementation of this design did not cause any problems, when programming this system to the board, the PowerPC did not produce its expected outputs. This was because the BMM file was not included in the synthesis, and thus the Block RAM modules for the program instruction and data were not initialized correctly. Eventually, a script file created by PlanAhead was modified to include the BMM file generated by XPS. Unfortunately, this created an error in

synthesis. The BMM file had to be updated to include the additional top-level hierarchy caused by importing the design into ISE. Once this was updated, the synthesis ran without any errors. The system was programmed to the board and ran as expected. In order to further modify the software, the full bit file and the updated BMM file generated by PlanAhead had to be imported back into the EDK project directory. When downloading the bitstream to the board in EDK, the full bit file generated by PlanAhead will be programmed, and the instruction and data will be properly initialized in the Block RAM modules.

The final step of the incremental design process was to use the PowerPC module to reprogram the partial bitstreams through the ICAP. The EDK project had to be re-synthesized with the opb\_hwicap IP core included in the design. Once this synthesis finished, the remainder of the tool flow was unchanged from the previously described incremental version of the design. Thus, the self-reconfigurable system was properly implemented. The exact tool flow to achieve this result will be detailed in the “Production Flow” Section of this report.

### 5.1.5 Internal Reconfiguration Access Port

The Internal Reconfiguration Access Port (ICAP) module is used to perform partial self-reconfiguration on Virtex-II and Virtex-II Pro FPGAs [8]. The ICAP module is not intended for full device configuration, but only for partial self-reconfiguration. Configuration data can be read from or written to the ICAP. The interface for the ICAP module is show in Figure 5-1 below. The ICAP uses byte-width input and output ports, which allows for faster reconfiguration than a serial interface. The maximum frequency at which the ICAP module can be clocked without using the BUSY signal is 66MHz [2].



**Figure 5-1– Interface for the ICAP module**

The low-level details of the interface to the ICAP module are hidden by the EDK OPB\_HwIcap module provided by Xilinx. This module includes the necessary VHDL wrapper files for interfacing the PowerPC to the ICAP as well as the C software libraries that provide the methods for the ICAP functionality. The software library includes functions that allow for initializing the ICAP, reading configuration data from the ICAP, writing a frame to the ICAP, and writing a partial bitstream to the ICAP. Since the partial bitstreams will be stored in memory for this application, the most useful function is for writing a complete partial bitstream to the ICAP in which a pointer to the start address in memory and the size of the bitstream must be passed as parameters. The software libraries and VHDL wrapper files will convert this data into an appropriate format and write it to the ICAP module.

## **5.2 Design**

The design of a working self-reconfiguring system is a crucial stage in the development of a complete self-healing system. A true self-healing system must be able to fix errors in its circuitry without any external interaction. This reprogramming will be handled by one of the PowerPC cores on the Virtex-II FPGA. Therefore, this design will make use of both hardware and software elements. All of the code and other necessary files for the implementation of this system can be found in Appendix A. A step-by-step production flow to create a self-reconfiguring system is given in section 5.3 of this report.

### **5.2.1 Design Intent**

The purpose of this initial self-reconfiguring design is to determine and validate the functionality of the ICAP module for partially reprogramming portions of an FPGA. The prior General Dynamics C4 Systems MQP project team was able to perform partial reconfiguration through the JTAG connection with an external PC, but they were never able to successfully reprogram portions of the FPGA using the ICAP. In order to achieve the final goal of a self-

healing system that can detect and gracefully recover from errors, it is necessary for the FPGA to be able to reprogram itself without any external PC or human interaction.

Thus, this self-reconfiguring design is a relatively simple proof-of-concept circuit that demonstrates the use of the ICAP module and the production flow necessary to achieve this system. The partial module in this system is one that uses four switches to control the corresponding 4 LEDs on the bottom row of the FPGA development board. A static module includes the PowerPC system generated through the Xilinx EDK tool and a simple module that controls the other four LEDs on the FPGA development board. During the synthesis and implementation flow, a complete bit file will be generated with the static logic and the LED-switching partial module. Also, the partial bit files for the LED-switching module and the blank module will also be generated. These two partial modules will be written to memory. When the first button on the development board is pressed, the PowerPC will detect the input and will program the blank partial module to the ICAP. When the second button is pressed, the LED-switching partial module will be programmed. Thus, once the bitstreams are stored in memory, no external PC is required for reprogramming portions of the FPGA.

## **5.2.2 EDK Hardware Considerations**

The Xilinx EDK tool is the first place to start when considering the PowerPC design portion of this project. In EDK, it is important to start off with a new project that conforms to the settings of the project board. Interfaces like RS-232, and SDRAM should be added for proper operation of the PowerPC core. Once your workspace has been created, basic demonstration source code should have been created in order to test the interfaces which were added to the project. One extra interface which must be included for self-reconfiguration is the hardware ICAP wrapper. This peripheral can be added by going into the IP Catalog in EDK and manually added it to the design. Once the IP core has been added, connect the peripheral to the OPB bus and set a 64k address range for the device to operate on. The address range recommended in this tool flow is 0x40200000 to 0x4020FFFF.

### 5.2.3 EDK Software Considerations

There are two tools which can be used for the software considerations of the PowerPC design. The traditional EDK XPS tool can be used for hardware/software design, or for extra software design functionality the SDK tool can be used. This SDK tool is an adapted version of Eclipse and provides the user friendly development environment and features of Eclipse. No matter what tool is used, proper libraries and header files must have been included for the use of the hardware ICAP. Fortunately, the peripheral associated with the ICAP includes header files and high-level functions in order to interface with the ICAP for self-reconfiguration. The header file which must be included for all ICAP design is “xhwicap.h”.

```
Status = XHwIcap_Initialize(&HwIcap, DeviceId, XHI_TARGET_DEVICEID);
if (Status == XST_DEVICE_IS_STARTED)
{
    Print("Device is already initialized.");
}
else if (Status != XST_SUCCESS)
{
    Print("Failed to initialize: %d\r\n", Status);
    return XST_FAILURE;
}
Print("ICAP initialized...\r\n");
```

**Figure 5-2 – ICAP Initialization Code**

Figure 5-2 above is the code required for the initialization of the ICAP. When the software calls the function XHwIcap\_Initialize(), the status of the operation is returned and analyzed. If the status is “XST\_DEVICE\_IS\_STARTED” then the ICAP is already initialized. Otherwise if the status is not “XST\_SUCCESS” then the initialization failed and a status error is printed. If the initialization is a success then the code continues without branching.

The XHwIcap\_Initialize() commands requires three parameters and returns a status code. The first parameter of this function is a pointer to the hwicap instance variable. The second parameter is the device id value associated to the hardware ICAP wrapper of the design. This value is obtained from the “xparameters.h” header file. The third parameter is the device id of the FPGA being used. In this case of this project the device id is XHI\_XC2VP30.

This main function written for this project is shown below in Figure 5-3. The name of this function is program\_icap. This function requires three parameters. The first parameter is a



pointer to a 32-bit integer memory location of the partial bitstream to be programmed. The second parameter is the size of this bitstream and the third parameter is a pointer to the ICAP instance variable to be used. At the end of this function, the call either returns 0 for success or XST\_FAILURE for any type of failure. In the beginning of the function the normal programming ICAP print statement is invoked. The next function call is the ICAP function call that performs all partial self-reconfiguration in one line. This function is called XHwIcap\_SetConfiguration() and requires three parameters. The first parameter requires a pointer to an ICAP instance variable, which in this case will just be the third parameter of the program\_icap function. The second parameter required is a pointer to the buffer which contains the partial bitstream to be programmed, which in this case is the first parameter of the program\_icap function. Finally the last parameter is the size of the partial bits stream about to be programmed, which in this case is the second parameter of the program\_icap function divided by four. The reason why this is divided by four is because the original file size is given in bytes and bitstream is given as a 32-bit integer buffer, therefore each 32-bit reference is four bytes. After the function has been invoked the status value is placed and checked by a conditional statement. If the status is not a success then the function will print failure with a status code and return error. Otherwise the software goes on to invoke XHwIcap\_CommandDesync(HwIcap) and prints that it successfully programmed the partial bitstream. The CommandDesync must be invoked at the end of any partial reprogramming. This was done in many different examples of ICAP programming, and the programming was found not to be finalized until this command was issued. Lastly, the software returns successfully out of the function call and back to the procedure of which it was called.

```

int program_icap(Xuint32 * bit_file_addr, int size, XHwIcap * HwIcap) {
    Print("programming ICAP...\r\n");

    XStatus Status = XHwIcap_SetConfiguration(HwIcap, bit_file_addr, (size/4));

    if (Status != XST_SUCCESS)
    {
        Print("Failed to program: %d\r\n", Status);
        return XST_FAILURE;
    }

    XHwIcap_CommandDesync(HwIcap);
    Print("Programmed successfully: %d\r\n", Status);

    return 0;
}

```

**Figure 5-3 – Main ICAP Programming Function**

## 5.2.4 Top-Level

The design of a partially reconfigurable system requires a hierarchical design approach to be followed. All global logic, including I/O and global clocks, must be included in the top-level module. Also, all top-level I/O for the system must be declared in the top-level file. Multiple static and reconfigurable modules can be instantiated in the top-level design module as black boxes. No synthesized logic should be included in the top-level file, so all designs must be part of a black box module. When using these black box modules, Xilinx synthesizes an empty module with the correct number of inputs and outputs. The individual modules must be synthesized in separate ISE projects. The top-level module must also include the bus macros for all non-global clock signals between the partial modules and the rest of the design. Only a clock signal may be connected to a reconfigurable module without passing through a bus macro. The bus macros are instantiated and used as normal entities, and the necessary ports are listed in the bus macro package file corresponding to the type of the FPGA being used. Bus macros are included to maintain proper I/O connection to the partially reconfigurable area regardless of what module is currently programmed there. Thus, they provide static routing for the reconfigurable module [7].

The synthesis of the top-level file also requires special considerations. All designs in this project were synthesized with Xilinx Synthesis Technology (XST) tool, although other synthesis tools may also be used. The synthesis tool must be configured so that no optimizations occur

across hierarchical boundaries and I/O buffers must not be inserted in the lower-level modules. When using ISE to synthesize the top-level module, the synthesis properties must first be edited to modify these settings. To access this window, first click on the top-level VHDL file in the “Sources” panel. Next, right click on “Synthesize – XST” in the “Processes” panel, and select “Properties.” The “Property display level” will have to be changed to “Advanced” to view these synthesis options. Under the “Synthesis Options” category, “**Keep Hierarchy**” must be set to “Yes”. Under the Xilinx Specific Options category, “**Add I/O Buffers**” must be enabled. The XST process properties window is shown in Figure 5-4 below with the correct settings. When using command-line synthesis or a different synthesis tool, similar directives must be set for the top-level module.

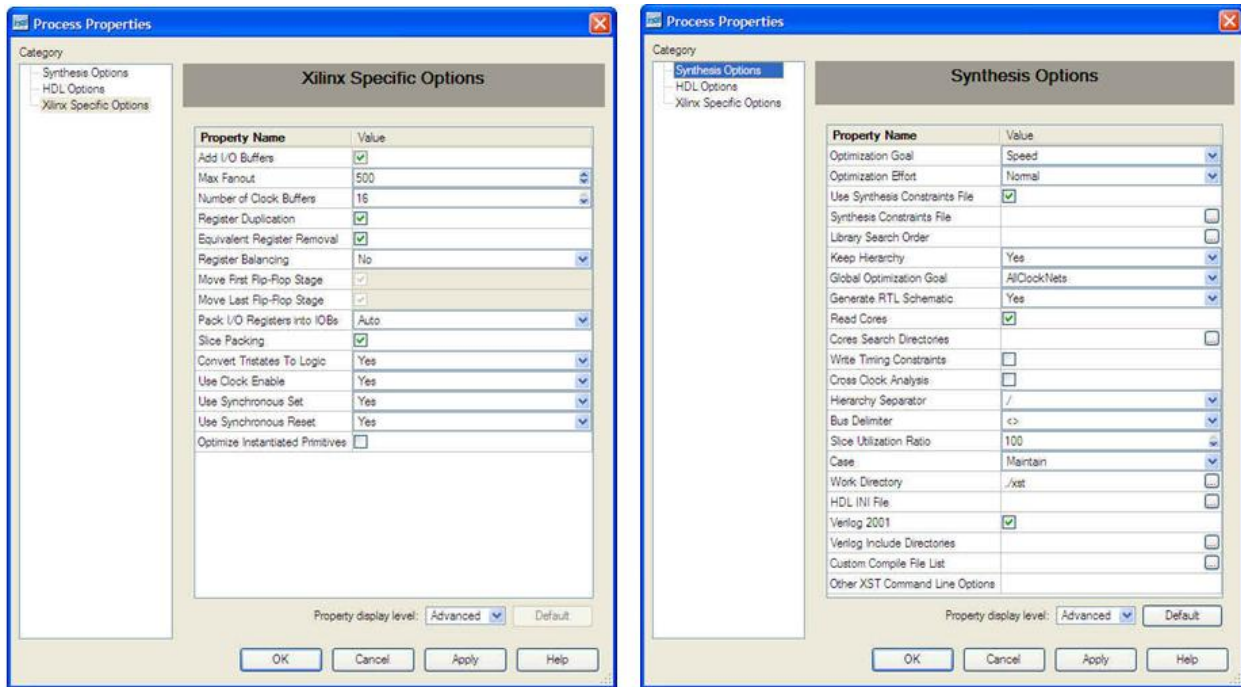


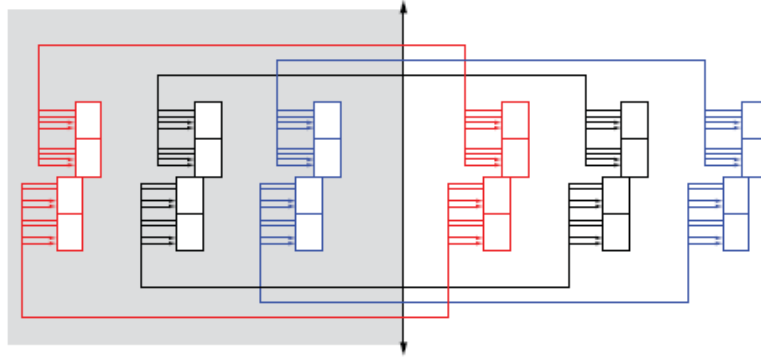
Figure 5-4 – XST Process Properties

The top-level module in this design includes the partially reconfigurable module for the LED-switching circuit that controls four output LEDs and a single static module that incorporates the PowerPC system and a module that controls the states of the remaining four LEDs on the development board. The necessary system I/O is also included in the top level file, and the necessary bus macros are instantiated and connected to properly communicate with the partial module.

## 5.2.5 Bus Macros

Bus macros are physical ports that connect a reconfigurable module with other modules in the design [6]. Bus macros are a crucial design requirement in systems implanting partial reconfiguration. All connections between the partially reconfigurable module and the rest of the design must pass through a bus macro, with the exception of the clock signal [7]. These eight-bit bus macros are specific to the architecture of the FPGA being used and are supplied through the Early Access PR Lounge [7]. Currently, access to this lounge is restricted, and an additional online registration is necessary. Bus macros must be instantiated in the top-level VHDL file using port mapping. Bus macros have .nmc file extensions, which are pre-placed, pre-routed hard macros.

Several different types of bus macros are available for each FPGA architecture. Bus macros are unidirectional and have a specific signal direction. For the Virtex-II/Pro FPGAs, the signal directions are left-to-right and right-to-left. Whether the bus macros act as an input or output to the reconfigurable module depends on its signal direction and its placement. For instance, a left-to-right bus macro placed on the left side of the reconfigurable module acts as an input, while the same bus macro placed on the right side acts as an output. Virtex-4 FPGAs also include top-to-bottom and bottom-to-top bus macros. Separate bus macros are also supplied for the desired width. A “wide” bus macro is four Configurable Logic Blocks (CLBs) wide, while a narrow bus macro is two CLBs wide. These widths refer to the physical width of the bus macro and not to the data bandwidth. Wide bus macros can be nested along a single CLB row, as shown in Figure 2-5. For more complex designs with many I/O pins on the reconfigurable modules, wide bus macros may be desirable to save vertical space. Bus macros can also be synchronous or asynchronous. Synchronous bus macros register signals passing through it and provide better timing performance. Xilinx recommends that they are used in designs that can accommodate the additional latency [7]. Lastly, bus macros provide an optional bus macro enable control signal. When the enable signal is deasserted (set to 0), the bus macro outputs are set to 0. Since output signals for a partially reconfigurable module are unpredictable during partial reconfiguration, it is recommended that the enable signal is deasserted before loading a partial bitstream and then asserted once the bitstream has been loaded.



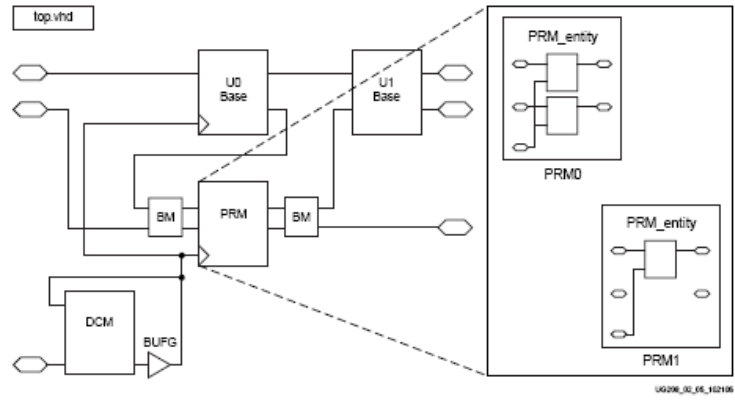
**Figure 5-5 – Nested Wide Bus Macros**

The bus macros used in all designs for this project are asynchronous, narrow bus macros without enable control. Although Xilinx recommends that synchronous bus macros with enable control are used, errors were encountered during synthesis when using these bus macros. There is unfortunately little documentation provided by Xilinx about the actual implementation details for using bus macros. Therefore, asynchronous bus macros without enable control were used to alleviate these errors. Since all of the designs for this project are relatively simple, no problems are encountered while using asynchronous bus macros. During initial testing, it appears that the output of the bus macros are 0 during reconfiguration even without using the enable signals. For added protection, though, manual enable control was implemented in VHDL to prevent erroneous outputs from reaching other modules or the output of the system. In this particular self-healing design, four bus macros were used, which provided an input and output bus macro on each side of the partially reconfigurable module.

## 5.2.6 Module Level

Module-level design in a partially reconfigurable system must also follow a few synthesis and design constraints. These lower-level modules should not contain any clock or reset primitives. The entities of all of the modules must match the corresponding port mapping performed in the top-level module. Also, all partial modules for a given partially reconfigurable region must have identical port definitions and entity names. This allows each of the partial modules for a given region to be linked from the same top-level design. Similarly, it is recommended that all partial modules for a given region have the same module name and file

name. Consequently, each version of a partially reconfigurable module should be saved in a different project folder to keep them separate. Figure 5-6 below demonstrates the matching port definitions and entity names for separate partial modules that are associated with the same partially reconfigurable module [7].



**Figure 5-6 – Matching Port Definitions and Entity Names for a Partial Module**

To import an EDK project into a static module, an existing source (not a copy) must be added to the ISE project. In the following dialog window, the \*.xmp file generated by EDK must be added to the ISE project. The I/O port information for the PowerPC system is given in the system\_stub.vhd file in the hdl directory in the EDK project.

Similar to the top-level synthesis, module-level synthesis should preserve hierarchy by setting the “**Keep Hierarchy**” synthesis option to “Yes.” As opposed to the top-level synthesis, though, all lower-level modules require that I/O buffers are not added during synthesis. Thus, the “**Add I/O Buffers**” synthesis option must be disabled [10].

The specific partially reconfigurable module implemented in this design is a simple circuit that assigns the values of the four-bit switch input to the four-bit LED output. It is important to note that any such assignments from an input to an output must be passed through a buffer or some other logic gate; otherwise, they will be optimized away during synthesis and an error will occur during later stages of the design implementation. The static module is made up of the LED driver for the remaining four LEDs on the development board and the PowerPC system generated by EDK. The LED driver is used to blink one LED at 2 Hz and another LED at 20 kHz. The remaining two LEDs are set to logic ‘0’ to permanently turn on the LEDs. This LED driver design was imported from the previous project team’s test circuit [10]. The

PowerPC module includes I/O necessary for communication with various parts of the Memec development board, especially the memory and RS-232 ports. The interface with the memory is necessary for reading the partial bitstreams from memory and programming them to the ICAP. The interface with the RS-232 port allows for status messages to be easily printed to a HyperTerminal window on a separate PC.

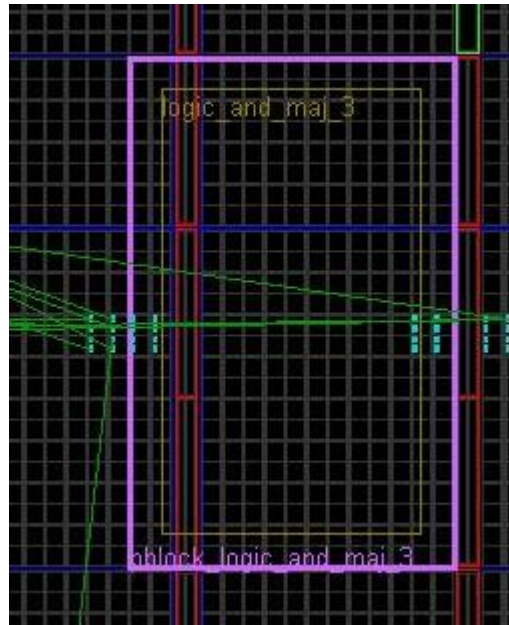
### 5.2.7 PlanAhead Implementation

Once the VHDL design and synthesis has been completed in Xilinx ISE, several net list (.ngc) files are created. These net list files are imported into PlanAhead, which is a floorplanning tool with support for partially reconfigurable modules. Once a new project is created in PlanAhead, a top-level net list file must be specified, and the remaining net list files may be imported by selecting the ISE project folders that include the necessary net list files. All static module net lists must be imported, but only a single reconfigurable implementation for a given reconfigurable module may be imported. Also, a user-constraints file (.ucf) should be imported at this step. Please note, though, that to enable partial reconfiguration support in PlanAhead, the following command must be entered into the console:

```
HDI::param set -name project.enablePR -bvalue yes
```

The budgeting and floorplanning of the static and reconfigurable modules must follow implementation rules, which are detailed in the “Dynamic Partial Reconfiguration of a Field Programmable Gate Array” report [10]. Once all of the partial modules are created, bus macros must be added to the floorplan. The bus macro .nmc files must be included in the PlanAhead floorplan directory; otherwise, the error message “ERROR:NgdBuild:630 - module 'bus\_macro\_instance\_name' is missing an AREA\_GROUP property” will be given during the NGDBuild portion of the production flow. Bus macros cannot be automatically placed by the synthesis or floorplanning tools, so the user must explicitly define where the bus macros are to be placed. Due to the unidirectional nature of bus macros, they must be placed on the correct side of the reconfigurable module for proper operation. It is useful to name bus macros by their direction, such as “inputbusleft” or “outputbusright.” To place the bus macros, PlanAhead must

be set to “Site Constraint Mode.” The bus macros are found in the primitives folder. To place a bus macro, select it from the primitives folder, and drag it onto the floorplanning view. It must be placed such that it straddles the boundary between the partially reconfigurable region and the rest of the board. The mouse must be hovering over the lower left corner of the slice to the left of the boundary. Once the rectangle representing the bus macro expands to be placed equally inside and outside of the reconfigurable module, the bus macro may be placed. For the eight-bit bus macros used in this design, eight ports will be placed outside of the reconfigurable region and eight corresponding ports will be placed inside the reconfigurable region. The proper bus macro placement for a reconfigurable module is shown in Figure 5-7 below.



**Figure 5-7 – Bus Macro Placement**

Once all of the necessary static modules, partial modules, and bus macros have been placed, the floorplanning stage of the design is complete. The specifics of the rest of the implementation are detailed in Section 5.3 – Production Flow. A new floorplan must first be exported before continuing with the design. PlanAhead is then used to generate script files specific to the PlanAhead design flow that will implement the rest of the design. These script files are generated by selecting “Run Partial Re-Config Flow” from the “Tools” menu. Once these scripts are generated, a small modification must be made to these scripts to properly initialize the Block RAM with instruction code for the PowerPC, as described in the Tool Flow



section of this report. Once this change is made, the buildAll.bat batch file may be run to complete the rest of the design process. This will generate the bit files for this design and place them in the “merge” directory of the PlanAhead floorplan folder. The static\_full.bit file is the complete design, with the partial module included in the partial region. The <partial\_module>\_cv\_routed\_partial.bit file is the partial bitstream for the partial module, where <partial\_module> is the name given to the partially reconfigurable area in PlanAhead. Similarly, the <partial\_module>\_blank.bit file is the blanking bitstream that will clear the corresponding partial module from the FPGA when programmed.

### 5.3 Production Flow

When creating a self-reconfigurable system there are many specific considerations to take. This production flow will cover all specific changes which must occur in all tool flows in order to integrate the PowerPC wrappers with any custom partial or static VHDL modules. Following these steps it should be possible to recreate the entire design. The following tool flow is best used with the exact versions of Xilinx tools described in Section 2.6 – Tools. This tool flow is not guaranteed with any older or newer version of these Xilinx tools.

- 1) Open EDK and create a new project with the project wizard. Specify your specific board settings. Add desirable peripherals. Set instructions to be stored in iocm\_cntlr and Data/Stack/Heap to docm\_cntlr.
- 2) Once the project has been created, go into the IP Catalog in EDK. Under FPGA Reconfiguration, double-click on opb\_hwicap, this will add the IP core to the system assembly view. Now connect ‘SOPB’ under opb\_hwicap\_0 to the OPB bus.
- 3) Double-click on opb\_hwicap\_0 and assign a 64k address space currently not in use. For example: 0x40200000 to 0x4020FFFF.
- 4) Perform “Software->Clean Libraries”, then perform “Software->Generate libraries and BSPs”, lastly perform “Device Configuration->Update Bitstream”. Depending on your computer, this process should roughly take 15 to 20 minutes.
- 5) Open ISE and create a new project with your board settings. Create a top level VHDL module. This module will include port mapping references to any static or partial designs.

- 6) Keep in mind, the static and partial modules to be linked into the top level should all comply with the design rules of the partial reconfiguration flow described in “Dynamic Partial Reconfiguration of a Field Programmable Gate Array”.
- 7) In your top-level ISE project add an existing source (not a copy). Import the \*.XMP file created in your EDK project into ISE.
- 8) Map all ports at the top level so that the EDK project, the static modules and the partial modules share all port constraints with no conflicts.
- 9) Locate the UCF file of the XPS project and add it as the top level UCF of the ISE project.
- 10) Add any new port mappings to the UCF for the other static/partial files if needed. Now double-click on implementation and stop the process once it processes and creates a new BMM file.
- 11) Open PlanAhead and create a new project. Import the top-level NGC file created during synthesis. Point reference to any folders which contain NGC files for child modules described within the top-level NGC. Select the correct part number for your board, import the UCF file created in ISE and click finish.
- 12) For every static module, right click on the static module and click create new pblock.
- 13) Manually input the follow command into PlanAhead’s command line:  
*HDI::param set -name project.enablePR -bvalue yes*
- 14) For every partial module, right click on the partial module and click on draw a pblock.
- 15) Size out the modules and place the bus macros according to the partial reconfiguration flow described in “Dynamic Partial Reconfiguration of a Field Programmable Gate Array”.
- 16) For every partial module on the floorplan, add the MODE attribute to it and set it to RECONFIG.
- 17) Next perform “File->Export Floorplan”. In the window, set the option to partial reconfig and click on finish.
- 18) Next perform “Tools-> Run Partial Reconfig”, In the window click on generate script files only and click on finish.
- 19) Go to the floorplan plan directory created and place the BMM file created from the ISE top-level project into the floorplan directory. Open the static folder and edit the

“staticlogicImpl.bat” file. Next, append the “-bm ..\bmmfilename.bmm” flag to the ngdbuild command.

- 20) Go back to the floorplan directory and execute BuildAll.bat.
- 21) After the script is complete, the merge folder should contain your full bit file, your partial bit file and your blank partial bit file.
- 22) Before programming the bit file to the board, ensure that the Mode Pins are set to Slave Serial Mode (M0 = 1, M1 = 1, M2 = 2). The Boundary Scan Mode disables the ICAP. JTAG configuration is still available, though, because it overrides other configuration modes [11].

### **5.3.1 Considerations for Further Development in EDK**

Once you finished the entire tool flow your full bit file should contain the PowerPC wrappers working in unison with any partial/static modules created in ISE. To continue further software development in EDK using this merged full bit file, follow the following tool flow:

- 1) In the merge folder, grab a copy of the full bit file created.
- 2) Place a copy of this bit file into the implementation folder in the EDK project directory.
- 3) Place a copy of the BMM file from your floorplan directory into the implementation folder in the EDK project directory.
- 4) Create backups of your current system.bit and edkBmmFile\_bd.bmm.
- 5) Rename your newly copied bit and bmm files to system.bit and edkBmmFile\_bd.bmm.
- 6) Perform any software build changes. Warning: Any hardware changes applied in your EDK project will result in a new synthesis overwriting your merged bit and bmm file.

### **5.3.2 Onboard Partial File Storage**

Once you have created all full and partial bit files and have the ability to write PowerPC code that can effectively be merged into the design, the next consideration is finding a place to store partial bit files onboard to be referenced by the ICAP. The most logical place to store this

data would be on some type of static ram or flash device. This project's implementation stores partial bit files on a 4MB flash. For the sake of simplicity the following tool flow assumes partial bit files of 70,000 bytes each.

- 1) Get a copy or copies of your partial bit file(s).
- 2) Open each partial bit file in a hex editor, and remove the header from the bitstream. The header is all of the data up to, but not including, the FF FF FF FF synchronization data. Once all of the data before FF FF FF FF has been deleted, save the modified bit files.
- 3) If you have more than one partial bit file open up a hex editor and append all bit files into one large partial bit file array. Keep note of the offset which specific partial bit files start and end.
- 4) Open up your EDK project and download your current bitstream to the board.
- 5) In the Device configuration menu click on Program Flash Memory.
- 6) Select the flash memory of your choice in the "Flash Memory Properties" portion of the window
- 7) Select the scratch pad memory of your choice in the "Scratch Memory Properties" portion of the window.
- 8) Do not select a bootloader and load the file you want to flash.
- 9) EDK should successfully flash the device with the data specified.
- 10) If the memory base address is 0x40000000 and you programmed three partial bit files 70,000 bytes each, your offsets should be:
  - Bank 1: 0x40000000-0x4001116F
  - Bank 2: 0x40011170-0x400222DF
  - Bank 3: 0x400222E0-0x4003344F

Once the partial bit files have been successfully programmed into flash memory, these partial bit files (or partial banks) can be referenced to the ICAP for partial reprogramming. Figure 5-8 below illustrates the memory mapping of these partial bit files in flash memory.

## Partial Bit Files Memory Mapping for the ICAP

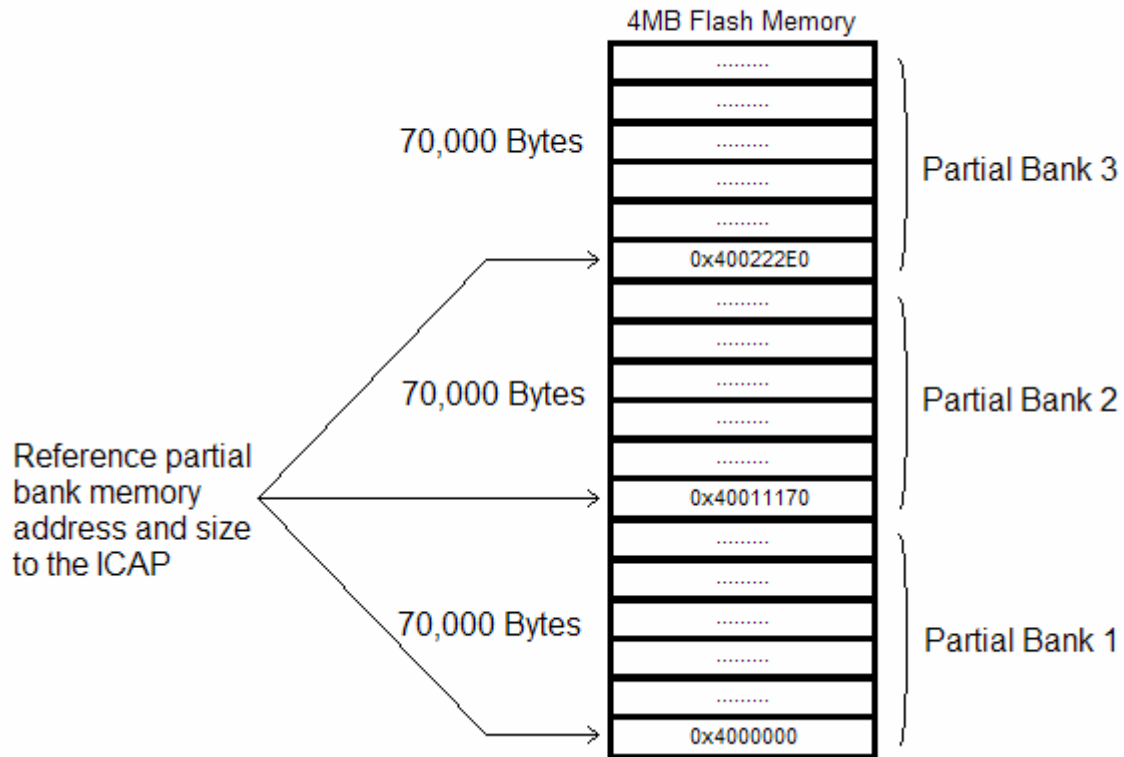


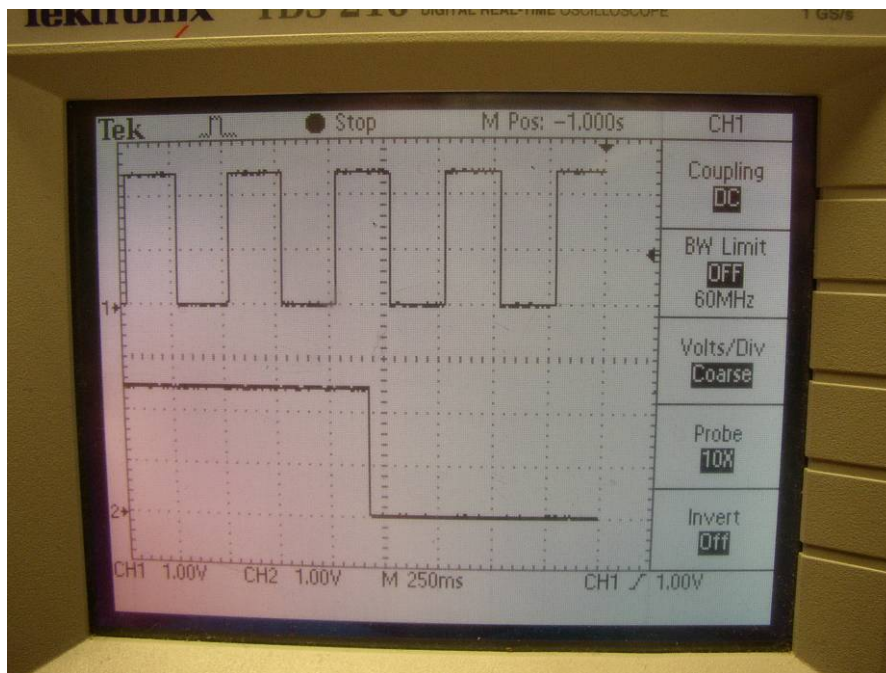
Figure 5-8 – Partial Bit Files Memory Mapping for the ICAP

### 5.4 Testing and Results

The proof-of-concept self-reconfiguring system uses the LED display as well as an RS-232 interface for easy visual testing purposes. The outputs of the LED modules can also be observed more closely with an oscilloscope. This is especially useful for monitoring the outputs of the static module while the partial module is being reprogrammed. The goal of this self-reconfiguring system is to reprogram a portion of the hardware without the need for an external computer and without affecting the performance of the remaining static hardware. The partially reconfigurable LED-switching module allows for the four LEDs to be controlled by four corresponding switches when the partial module is programmed and for none of the four LEDs to be active when the blank module is programmed. A successful self-reconfiguring system will allow for the partially reconfigurable module to be reprogrammed by a button push to either the

LED-switching module or the blank module without affecting the static module outputs, specifically the four remaining LEDs and RS-232 port communication.

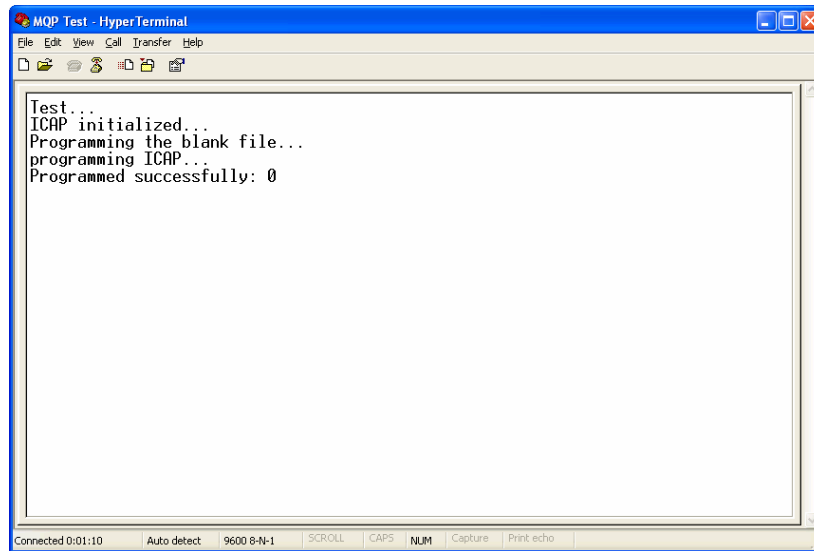
When the full bit file is first programmed to the board over the JTAG interface, the LED-switching module and all static logic outputs perform as expected. The four leftmost switches on the development board control the bottom four LED outputs from the partial LED-switching module. The static module successfully controls the top four LED outputs. The first LED blinks at 2 Hz, the second LED blinks at 20 kHz, and the remaining two LEDs are always on. Also, the HyperTerminal window on the PC displays the message “ICAP initialized...”



**Figure 5-9 – Oscilloscope Reading of 2 Hz LED during Switching to Blank Reconfiguration**

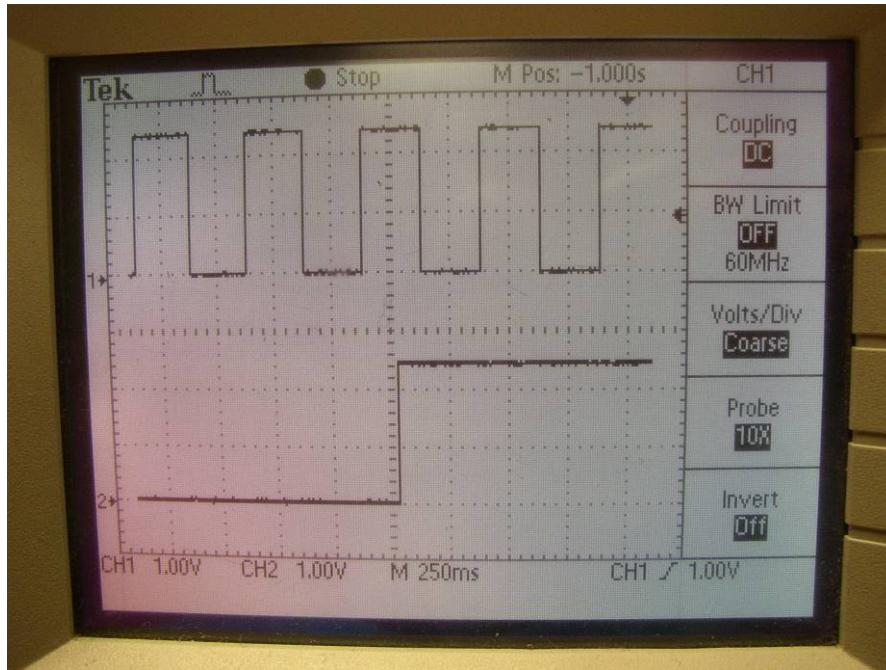
Once the two partial bit files are programmed to memory, the rest of the system is observed to work correctly. When the leftmost button on the development board is pressed, the blank partial module is programmed to the ICAP. None of the bottom row of LEDs controlled by the partial module are lit. The static module continuously drives the LEDs without any glitches resulting from the reprogramming. This functionality is verified by the oscilloscope reading shown in Figure 5-9 above. As shown in this image, the 2 Hz blinking LED output does not glitch during the reprogramming of the partial module. Also, the HyperTerminal window displays the message “Programming the blank file...” followed by the message “Programmed successfully: 0,” as shown in Figure 5-10 below. These status messages indicate that the button

press was detected by the PowerPC, and the corresponding blank module in memory was successfully programmed to the ICAP.



**Figure 5-10 – HyperTerminal Output for Blank Reconfiguration**

Similar to the previous test of reprogramming the LED-switching module to the blank module, a similar test was performed for reprogramming the blank module to the LED-switching module. This occurs when the second button is pressed. The LEDs controlled by the partial module are lit based on the leftmost four switches. Once again, the LEDs controlled by the static module do not glitch during the reprogramming of the partial module. This functionality is verified by the oscilloscope reading shown in Figure 5-11 below. As shown in this image, the 2 Hz blinking LED output remains valid during the reprogramming of the partial module. The HyperTerminal window displays the message “Programming the partial file...” followed by the message “Programmed successfully: 0,” as shown in Figure 5-12 below. Thus, the PowerPC successfully programmed the LED-switching partial module from memory to the ICAP.



**Figure 5-11 – Oscilloscope Reading of 2 Hz LED during Blank to Switching Reconfiguration**

```

MQP Test - HyperTerminal
File Edit View Call Transfer Help
Test...
ICAP initialized...
Programming the blank file...
programming ICAP...
Programmed successfully: 0
Programming the partial file...
programming ICAP...
Programmed successfully: 0

```

Connected 0:01:10 | Auto detect | 9600 8-N-1 | SCROLL | CAPS | NUM | Capture | Print echo

**Figure 5-12 – HyperTerminal Output for LED-Switching Reconfiguration**

The proof-of-concept self-reconfiguring circuit passed all of the desired tests to prove the functionality of the ICAP for partially reconfiguring a module. The blank and partial bit files were loaded to the ICAP multiple times in random order, and the system demonstrated the proper behavior in all scenarios. The four LEDs controlled by the partial module were set to the corresponding switch values when the partial file was programmed and were off when the blank file was programmed. The static circuitry, including the module to control the other 4 LEDs and



the PowerPC, remained fully functional without any glitches during the reprogramming of the partial module. Thus, the concept of the self-reconfiguring circuitry designed in this portion of the project may now be used as a base design to develop a complete self-healing circuit that can be used detect and gracefully recover from errors.

## 6 Implementation of a TMR-Based Self-Healing System

The self-reconfiguring system implemented in the previous chapter verified the capability of using the ICAP module to successfully reprogram a portion of an FPGA without an external PC. The successful design of a self-reconfiguring system is a crucial accomplishment in the goal of a true self-healing system. However, the current implementation of a self-reconfiguring system still requires human interaction to reprogram a partially reconfigurable module based on a button press. Ultimately, the final goal for achieving a self-healing system is to integrate a self-reconfiguring system into a design using TMR. Such a design will replicate the desired module to be kept safe, and all outputs from this module will be compared to determine the majority value. Thus, if one of the replicated modules has an error, the majority output will still be valid and the module in error will be automatically reprogrammed.

### 6.1 Analysis

The implementation of a self-healing system requires significant design choices to be made regarding the application of TMR to a self-reconfiguring system. Many of the same considerations from the analysis of the self-reconfiguring system remain for the implementation of a self-healing TMR system. These considerations will not be repeated in this section, but only new design choices will be discussed. When using TMR in self-healing system, careful assessments must be made regarding exactly which modules are to be protected. There are many intricacies in a TMR design that increase the complexity of the system. Without careful analysis, though, vulnerabilities may remain in the system that could cause irreparable damage to the circuitry that cannot be fixed without stopping operation and reprogramming the entire board.

#### 6.1.1 Custom VHDL TMR Implementation vs. TMRTool

Xilinx provides a TMRTool that can be used with any synthesis tool to implement XTMR (Xilinx TMR) into a system. XTMR refers to the TMR approach explained in section 2.3 – Triple Module Redundancy where the voting circuitry is triply replicated along with the

modules. The use of the TMRTTool reduces errors in TMR designs and is more scalable to future designs than a custom VHDL implementation [19]. The TMRTTool also helps the designer to focus on the logic of the system rather than the logic necessary for the TMR implementation. The TMR implementation resulting from the TMRTTool may also be more optimized than a custom solution and should provide complete SEU and SET immunity [15].

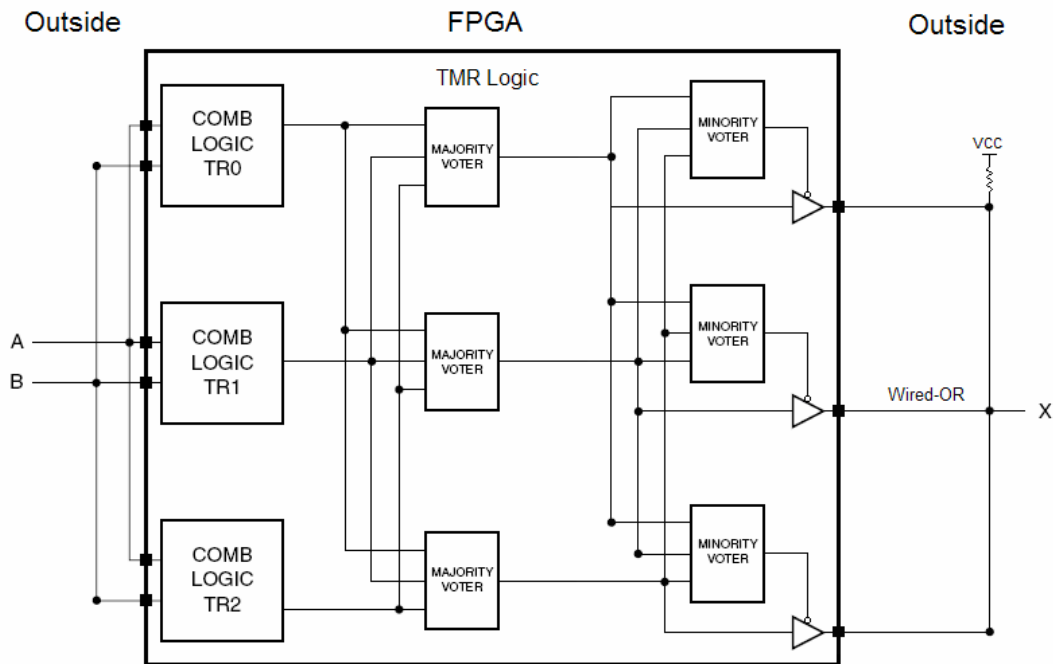
Although the Xilinx TMRTTool may result in a more efficient TMR implementation than custom VHDL, several drawbacks opposed the use of this tool. In a prior project at General Dynamics C4 Systems, the TMRTTool was used on a proof-of-concept simple system. Despite Xilinx's claims that the TMRTTool is easy to use and will save the developer time over a custom implementation, this report found that it created a complex tool flow that was difficult to manage even in a relatively simple system [12]. Considering the already complicated tool flow and design requirements necessary for partial reconfiguration, the additional layer of complexity introduced by the TMRTTool is not desirable. Also, the use of the TMRTTool requires an additional product registration that was not currently available by General Dynamics C4 Systems. Therefore, the custom VHDL solution was chosen as the method of implementing TMR in this system. The additional design time spent creating the voting circuitry and interconnections was decided to be more worthwhile than the time that would be spent integrating the TMRTTool into the already complex tool flow. The custom VHDL solution will be based off of the XTMR approach, with the replicated voting circuits for additional security. Portions of the XTMR design had to be modified to be capable of being designed in VHDL. The specifics of the design will be outlined in section 6.2 of this chapter. Although the custom VHDL solution may not provide the same level of security and efficiency as the Xilinx TMRTTool, it does provide enough functionality for the development and testing of a proof-of-concept self-healing system.

### **6.1.2 Partial Regions and Safe Regions**

On the FPGA floorplan there are two regions of distinction to be made, the partial regions and the safe regions. The difference between the two regions is that the partial regions have the ability to be partially reprogrammed in order to mitigate any errors. The safe regions are regions that cannot be partially reprogrammed and for the sake of this project are assumed to be safe from any type of errors. In order to continue with the experiment decisions had to be made on

what exact logic elements should be added to the partial regions and what should stay in the safe regions. Also, the number of individual partial regions was a concern in the analysis. Once these details were sorted out, the design of all the logic modules could continue.

In this design, it was imperative to have the primary triple redundant modules in the partial region. These modules were the main concern and any errors which occur should be alleviated by means of partial reconfiguration. Note back to the Xilinx TMR implementation in Figure 6-1 below. Apart from the main logic module being triple redundant, the majority voters themselves are also triple redundant. These modules should also be partially reconfigured if a minority voter detects that it is abnormal. The main question here was which region the minority voter should be placed. One of the reasons why it should not be placed in the partial region is because of the fact that there is no error checking circuitry in place for it. If the minority voters faced any critical errors, there would be no means of determining this. This is one of the limitations of triple module redundancy. Regardless of how many layers of voting circuitry are added, the last layer will always be unchecked and therefore can be the point of failure. Although the minority voter was unchecked, the decision was made to keep the minority voter in the partial region. This decision was made to give the designer the ability to randomly scrub the entire partial region. If the minority voter contained a critical error the TMR circuitry may not be able to detect it, but placing it in the partial region will allow for periodic updates of the module with no system downtime or affect on the output. The modules that were kept out of the partial regions and assumed safe were the PowerPC wrappers, the TMR OR-gate output and the bus macros. The PowerPC wrappers and the bus macros may be able to be partially reprogrammed, but that functionality would be out of the scope of the self-healing experiment. The TMR OR-gate is the point of convergence for all three redundant outputs. This module was placed in the safe region because in an actual TMR implementation, the three outputs would be placed through a wired OR-gate off the FPGA as shown below in Figure 6-1.



**Figure 6-1 – Full TMR Logic Implementation**

Determining the number of partial regions was a major decision in our design. The design needed enough partial regions in order to have a seamless valid output during any reconfiguration, but could not contain too many partial regions in order to reduce bus macro complexity and length of synthesis. The design required at least three partial regions. In order for the TMR system to keep a valid output throughout reconfiguration, it requires at least two modules active at any given point. In order to be able to correct errors in any module and have the two other modules be active during reconfiguration, it was apparent that the three modules needed to have their own independent partial regions. The modules that were left are the majority and minority voters. The design could contain nine different partial regions for the three modules, majority voters and minority voters. Since the goal of the design is to have as few partial regions as possible, placing the majority and minority voters in the existing partial regions could be a possibility. Furthermore, for a given module during reconfiguration, the corresponding majority and minority voter for that module was not essential for the TMR system to function. Therefore, placing these modules in the partial region with their corresponding logic module in test is a safe design choice for the TMR implementation. The partial boundaries of this design are illustrated below in Figure 6-2.

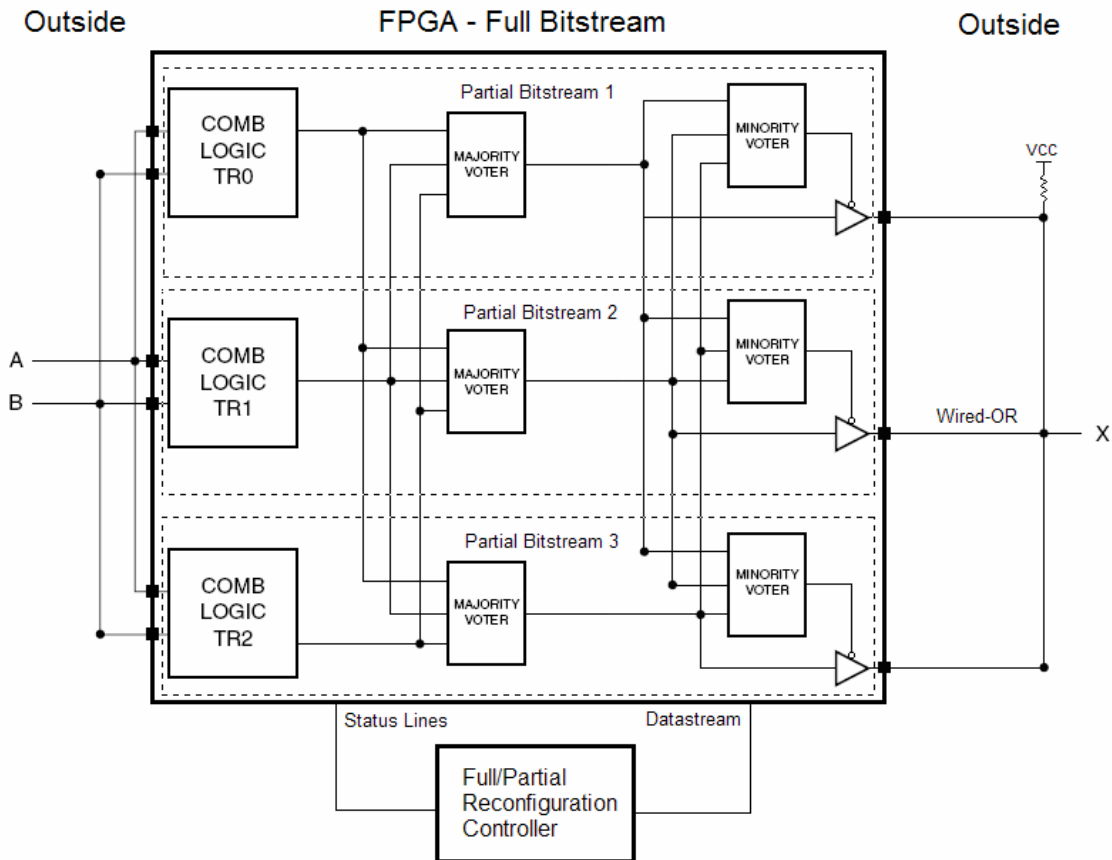


Figure 6-2 – Partial Region Boundaries

### 6.1.3 Error Decoding

Error decoding was analyzed in order to determine which partial region was placed in error at any given moment. To determine whether an error was placed some form of decoding must occur. This decoding must interpret the outputs of the redundant modules and the outputs of the minority voters. The outputs of the redundant modules must be decoded in order to determine which module is in error. The outputs of the minority voters must be decoded in order to detect which majority voter module is in error. The question is where this decoding should take place, and how this decoding can be made aware to the PowerPC. An initial thought was to perform the decoding in the voting circuitry. The drawback of this was that it adds more complexity to the voting circuitry. In either case, there still must be some form of bridging between the PowerPC and the custom VHDL modules. The solution to this problem was to create a custom peripheral

in EDK. Fortunately, Xilinx provides helpful sample VHDL code when creating a new custom peripheral. This peripheral contains register values and will have input from the minority voters and redundant modules. The peripheral also includes reset and enable outputs. It was decided that the peripheral would be the main source of error decoding for the design. Keeping error decoding within the peripheral adds simplicity to the voting logic implementation.

Since the circuit in this project is a 1-bit output this makes error decoding fairly simple. Once more outputs are introduced to the design, the error decoding becomes slightly more complex. In the current wrapper design, there are 6 bits of input being read and decoded. Three inputs are read from the redundant modules, and the other three are read from the minority voting modules. If any of these status signals differ from the other two, the slave register in the wrapper VHDL is set to an error state. Once the software in the PowerPC polls to read this register value, it will interpret the error state and know which partial module needs to be reconfigured. After the PowerPC reconfigures the area in question, it sets the slave register back to a normal state in order to decode any future errors.

#### **6.1.4 Error Sampling**

One careful consideration made was the rate of error sampling. In a TMR design there are three redundant modules which must read an input and the voting circuitry will test the output. The custom peripheral also tests this output to detect any errors with the module itself. However, if the sampling rate is too fast, the custom peripheral may violate timing constraints with the three modules. The current clock of the PowerPC is set at 100 MHz. This clock was initially used as the error sampling rate for our custom peripheral. However, it was found that the custom peripheral was reading incorrect values from the redundant modules and therefore profiling valid modules as having errors. To correct this, the timing constraints must satisfy the redundant modules. Extra time must be given to the modules to account for propagation delay. To do this, the clock rate of the custom peripheral sampling must be down converted to a slower clock rate. Once the slower sampling was achieved the modules had ample time to account for propagation delay, and the custom peripheral no longer flagged any false-positive readings on the output.

### **6.1.5 Output Enables**

As mentioned before, bus macros are the bridge between partial modules and any other types of modules. One issue arises when analyzing the state of an output when a partial module is being reconfigured. In the TMR design, all outputs at the end are placed through an OR gate to determine the true TMR output. However, it was found that when reconfiguring a partial region, one of the inputs of this OR gate was placed in an unknown value. From this test it was clear that some form of output enable must be introduced into the TMR design. This output enable is a simple AND gate that controls outputs of the partial region. This output enable is controlled by the PowerPC. Once the PowerPC is reconfiguring a partial region, it places all outputs from that partial region to logic '0'. The output enabling alleviated the unknown state issue at the output of the TMR system. From studying the different types of bus macros available, it was discovered that some bus macros already contained output enable signals for their outputs. When these bus macros were used in the design, it introduced compatibility errors and synthesis was not possible. Instead of looking into this matter, it was safer to revert to the bus macros which were compatible to the design.

### **6.1.6 PowerPC Partial Module Reset**

Another issue found was that after the PowerPC reconfigures a partial boundary, it does not reset the circuitry it reconfigured. Having the modules reset was important to the design, because it ensured that the modules are placed in a known state. To combat this issue, reset outputs were given to the custom peripheral. After the PowerPC detects an error and reconfigures the module in question, it resets the module after reconfiguration so that it is not kept in an unknown state. This issue only applied to state machines that included a reset input. Xilinx did not have the partial modules reset after reconfiguration in hopes to preserve state before and after reconfiguration. This is not helpful to the TMR design because during reconfiguration the other two modules could switch state while the third module is being reconfigured [18].



### **6.1.7 DCM Module Placement**

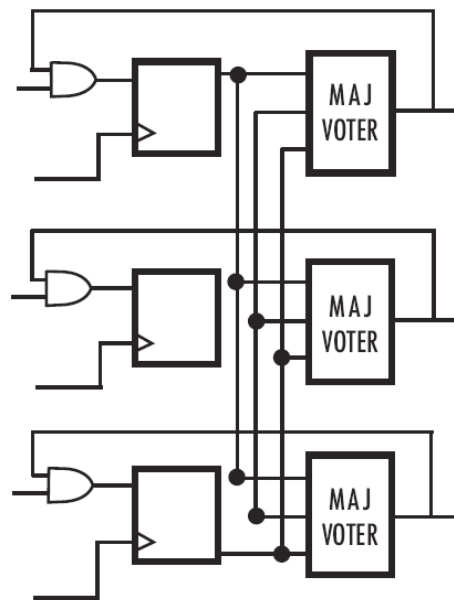
As a default in all EDK designs clock primitives such as DCMs and BUFGs are placed as wrappers under the EDK project tree. Issues arose when introducing state machines into the TMR system design. The design would not correctly build if the DCM was a wrapper under the EDK project hierarchy. Xilinx explains that having clock primitives in the EDK level of a partial reconfiguration design provides complications [7]. Suggestions were made to remove the DCM wrapper from the EDK project and instantiate it at the VHDL top-level design file. Following the recommendations, the DCM module was removed and placed at the PowerPC system level of the design. In doing this modification, the project was able to successfully build with no errors.

### **6.1.8 Considerations for Sequential Systems**

Using sequential state machines as the module to be replicated in a TMR design adds additional considerations for the synchronization of the three modules. In a combinational system, once a partial module is reconfigured, the output will be valid after a relatively short propagation delay through the entire logic module. Combinational systems do not have any need to be synchronized; only the logic must be reprogrammed. A sequential system, however, uses registers and feedback to create a state-machine. A registered output or next-state logic value provides the feedback to determine which state will be entered at the next clock cycle based on the current state. If the current state is invalid, the next state will likely be invalid as well since the next state logic depends on the current state. Therefore, synchronous systems require additional considerations for synchronization between the three replicated modules. Once a module is reconfigured, it must be synchronized with the other two modules before its outputs can be relied upon in the TMR voting circuitry. If the state is not synchronized correctly, or if the PowerPC checks for an error before the synchronization occurs, the partial module may be reconfigured again, which could potentially result in an endless reconfiguration loop if the states are not synchronized correctly.

One possible solution to this problem is to reset the all state machines in the design after a module is partially reconfigured [4]. This solution is not ideal, however, because the operation of the circuit must be stopped to reset all of the state machines, even those that had previously

been working. A self-healing system should be able to reprogram any modules in error without affecting the rest of the system. The outputs should remain fully functional during and after a partial reconfiguration, which would not be possible if all state machines had to be reprogrammed in order to achieve synchronization. Another potential solution is to implement a synchronization state in the state machine that allows for working modules to re-synchronize the newly reconfigured module. Of the two working modules, it may be possible to use one module to re-synchronize the reconfigured module while the other module acts as the sole output of the system until all state machines are synchronized. A design implementing this method would require substantial analysis and debugging, though, which was not practical in the scope of this project.



**Figure 6-3 – Voting on Feedback for Redundant State Machines**

The synchronizing solution used for the implementation of this self-healing system is to insert voters on all of the feedback paths, as shown in Figure 6-3 above [19]. This design was again used from the XTMR implementation, although for this particular system it will be designed in custom VHDL. An alternative to using the majority voter output as feedback, the final voter output after the minority voters can also be used. This actually provides additional error checking since there could potentially be an error in a single majority voter module, which could cause the feedback to be erroneous. Since the next-state of each state machine will now be the majority-voted value, the state machines should stay synchronized, even after one is

reconfigured. Even if a module has the wrong current state, the majority next state being fed back to the module should cause it to enter into the same state as the other two modules, thereby synchronizing the system. It is important to note, that a small delay may be necessary for the synchronization to complete, so the PowerPC should be programmed to wait before checking for errors again.

### **6.1.9 Expandability of the TMR Design**

Although the goal of this portion of the project is to create a proof-of-concept self-healing system with a relatively simple logic module, considerations for future reusability and expandability of this project had to be made. This is especially true for the VHDL TMR implementation created for this project. The self-healing top-level module, as shown in Figure 6-2, demonstrates the complexity of the interconnections between the partially reconfigurable modules in this design. For each partial region containing a logic module, a majority voting module, and a minority voting module, inputs are required for the outputs of the other two logic modules and two majority voters. Correspondingly, each module must have outputs for each of its majority, minority, logic, and final voter outputs. The reconfigurable modules also must include inputs for any necessary logic, reset, clock, and enable inputs. Since all I/O between a partial module and the remainder of the design must pass through bus macros, with the exception of any clock signals, the top-level module in self-healing systems becomes very complex as additional outputs are added to the module. In fact, communication between two partial modules, such as the logic and voter outputs, must pass through the output bus macro of the first module and the input bus macro of the second module.

For the first version of a self-healing system, only the necessary bus macros for a single output were coded in the design. However, this leaves little room for expandability without having to re-code much of the top-level design. To alleviate this concern, a generic value NUM\_TMR was used to generate the correct number of TMR voting circuit modules and bus macros. Within each partial region, the number of majority and minority voters that are instantiated is defined by the NUM\_TMR generic value. The only modification that must be made to increase the number of voted outputs is to map the corresponding output of the logic module. All of the voters and interconnections are instantiated automatically, and the size of the

partial module I/O busses relating to the TMR voting circuitry is defined by the generic value. At the top-level module, the number of bus macros instantiated for I/O for each partially reconfigurable module is defined by the generic value. When modifying this design, only the logic, clock, and reset inputs must be modified as necessary in one section of the code. The remainder of the bus macro instantiation should work correctly for all interconnects between the modules and any output logic and voter values. Thus, only overall system I/O must be modified at the top-level module for bus macro communication with the partial modules. All interconnections and outputs of these modules are handled automatically. Once the design enters the floorplanning state in PlanAhead, all bus macros must be placed at the boundary of the corresponding partially reconfigurable region. Also, the custom EDK wrapper used to decode the error signals and inform the PowerPC which module needs to be reprogrammed is also made generic. This wrapper reads the logic and minority voter outputs as generically defined busses, and loops through all of the data to check for incorrect values which would indicate that a module is in error. This generic value can be set through the Xilinx Platform Studio software to match the NUM\_TMR value of the rest of the design.

Although the VHDL and EDK TMR implementation can be made to generically account for proper voting of any number of desired outputs, one portion of the design that cannot be made easily reusable is the feedback logic necessary to keep the replicated modules of sequential systems synchronized. As previously discussed, the feedback signals for state machines must be voted upon in order to keep all of the replicated modules synchronized. However, each state machine must be designed to handle this feedback logic uniquely. Some simple state machines can use the feedback logic as the only source of next-state logic for the modules. More complex state machines that include internal memory of values may require additional synchronization states or some other such consideration to keep the state machines synchronized after a partial reconfiguration. Thus, the feedback logic for each state machine must be designed uniquely for that machine.

## 6.2 Design

The implementation of a self-healing system involves a much more complicated design than that of a self-reconfiguring system. Many of the design topics discussed in section 5.2 must also be applied to the design of a self-healing system, but they will not be replicated in this section. In order to design a self-healing system, three partially reconfigurable modules must be included in the design instead of a single module, as was implemented in the self-reconfiguring system. Also, a self-healing system requires substantially more logic design as a result of the majority and minority voters required for the TMR implementation. A custom PowerPC wrapper must be created to decode the signals from the TMR circuitry to determine which modules are in error, and alert the PowerPC which module needs to be reconfigured. The top-level module also requires significantly more design work as a result of the bus macros needed for TMR inter-module communication.

### 6.2.1 Design Intent

The successful design and implementation of a self-healing system would satisfy the overall goal of this project. The previous design of a self-reconfiguring system will act as a baseline for the design of a self-healing system. Now that the FPGA can partially reprogram itself using the ICAP interface, an additional level of TMR circuitry will allow the system to detect and reconfigure any modules in error while maintaining valid outputs from the system.

The self-healing system designed for this project is a proof-of-concept circuit that demonstrates the use of partial self-reconfiguration and TMR to detect and gracefully recover from errors in a system. The production flow necessary to replicate such a system is given in the following section 6.3. The module being replicated in this system blinks an LED at a rate of 1 Hz. When a button is pushed on the Memec development board, the rate of blinking is increased to 8 Hz until the module is reconfigured. This module, along with the necessary majority and minority voters, is replicated three times for the TMR implementation. The three buttons on the development board cause errors on the corresponding three replicated modules. The logic output of each module as well as the minority voter output for that module are displayed on the leftmost six LEDs of the board (the top three are the logic outputs, the bottom three are the corresponding

minority voter outputs). The bottom-rightmost LED displays the final output of the system, which should remain at 1 Hz even if a module is currently in error.

## 6.2.2 EDK Software Design

The software of the PowerPC is an important part of the entire self-healing system. The software reacts to errors flagged by the TMR system in hardware and acts on them accordingly. Appendix B contains the C code written for the PowerPC that fully implements partial reconfiguration management. The portion of code below in Figure 6-4 is the beginning of the main procedure. Similar to the code in Figure 5-2, the software initializes the ICAP peripheral and displays the status if the ICAP was successful in initialization or failure if the ICAP failed to initialize.

```
int main() {
    Xuint32 baseaddr = 0x42000000;
    Xuint32 Reg32Value;
    XStatus Status;
    int i;

    Status = XHwIcap_Initialize(&HwIcap, HWICAP_DEVICEID, XHI_TARGET_DEVICEID);
    if (Status == XST_DEVICE_IS_STARTED)
    {
        xil_printf("Device is already initialized.");
    }
    else if (Status != XST_SUCCESS)
    {
        xil_printf("Failed to initialize: %d\r\n", Status);
        return 1;
    }
    xil_printf("ICAP initialized...\r\n");

    xil_printf("Starting tmr_wrapper_test...\r\n");
    TMR_PPC_mWriteSlaveReg0(baseaddr, 0);
    TMR_PPC_mWriteSlaveReg1(baseaddr, 0);
    banner();
}
```

**Figure 6-4 – Beginning of main**

After the ICAP has initialized, slave register 0 and slave register 1 get initialized to 0 and the banner() procedure is called. The banner procedure is a procedure which prints a “next page” character to stdout in order to clear the HyperTerminal screen and displays the details of the number of TMR errors found. This procedure is shown below in Figure 6-5.

```

void banner() {
    outbyte(0x0C);
    print("Partial Reconfigurable TMR Monitor\r\n\r\nDeveloped by: Evan
Custodio and Brian Marsland\r\n\r\n");
    xil_printf("Total Number Of Bank A Errors: %d\r\n", bankA);
    xil_printf("Total Number Of Bank B Errors: %d\r\n", bankB);
    xil_printf("Total Number Of Bank C Errors: %d\r\n\r\n", bankC);
}

```

**Figure 6-5 – The banner procedure**

The next portion of the code is best shown in its entirety. This code is the remainder of the software and is shown below in Figure 6-6. In the beginning of this infinite loop is a for-loop which iterates from 10 to 0. This for-loop contains a software delay in the beginning which delays for less than a second. For each iteration in the loop the software reprints the string: “Next Iteration Check In: %d” where %d is the index of the loop. This loop’s main function is to count to from 10 to 0 showing the user when the next error checking will take place. The printing of character 0x0D forces a carriage return and allows for reprinting of the line.

After the for-loop is the main error checking code: “Reg32Value = TMR\_PPC\_mReadSlaveReg0(baseaddr);”. This line of code reads the register which contains error decoding information from the custom peripheral module. Depending on which value is placed in slave register 0 is what decides which conditional segment is executed. In the case of Reg32Value being 1, 2 or 3, similar functionality occurs. The global value for the corresponding partial bank is incremented for statistic purposes. The banner procedure is displayed to show these statistics. The error found and reprogramming statement is printed to the HyperTerminal screen. The corresponding output enable signals are disabled for the corresponding module before and during reconfiguration. The program\_icap function from the previous section is called upon the corresponding region in error. After reprogramming, the reset signal is set for that module and outputs are re-enabled. Lastly, slave register 0 which contains the error status is set back to 0 and the infinite loop then loops over. In the conditional statement, if there is no error set then the else segment is executed. In this segment of code the banner procedure is redisplayed and the statement: “No Errors Found In This Iteration Check!” is printed to the HyperTerminal screen.

Slave register 0 is the register which holds all the error information. Once the error is decoded at the custom peripheral then slave register 0 can get set to three different values 1,2 or 3. Value 1 corresponds to an error existing in partial bank A, Value 2 corresponds to an error

existing in partial bank B and Value 3 corresponds to an error existing in partial bank C. Slave register 1 is the register which controls all reset and output enable signals to the partial modules. The 7 least significant bits in the register connect to a reset or output enable for a specific partial module. Bits 1, 2 and 3 are connected to the reset signals of partial bank A, B and C respectively. Bits 5, 6 and 7 are connected to output enabled signals of partial bank A, B and C respectively. These bits were not arbitrarily chosen, these bits were chosen in order to easily set the signals in hexadecimal. Bits 1, 2 and 3 would be set with value 0x01, 0x02, and 0x04 and bits 5, 6 and 7 would be set with value 0x10, 0x20 and 0x40.



```

while (1) {
    for (i = 10; i > -1; i--) {
        delay(3000000*5);
        xil_printf("Next Iteration Check In: %d ", i);
        outbyte(0x0D);
    }

    Reg32Value = TMR_PPC_mReadSlaveReg0(baseaddr);

    if (Reg32Value == 1) {
        bankA++;
        banner();
        xil_printf("Error Found On Bank A!\r\n\r\n");
        xil_printf("Reprogramming Partial Bank A...\r\n");
        TMR_PPC_mWriteSlaveReg1(0x42000000, 0x40);
        program_icap((Xuint32 *)0x20400000,73824,&HwIcap);
        assert_reset(4);
        delay(2000000);
        TMR_PPC_mWriteSlaveReg0(baseaddr, 0);
    }
    else if (Reg32Value == 2) {
        bankB++;
        banner();
        xil_printf("Error Found On Bank B!\r\n\r\n");
        xil_printf("Reprogramming Partial Bank B...\r\n");
        TMR_PPC_mWriteSlaveReg1(0x42000000, 0x20);
        program_icap((Xuint32 *)0x20411C78,74236,&HwIcap);
        assert_reset(2);
        delay(2000000);
        TMR_PPC_mWriteSlaveReg0(baseaddr, 0);
    }
    else if (Reg32Value == 3) {
        bankC++;
        banner();
        xil_printf("Error Found On Bank C!\r\n\r\n");
        xil_printf("Reprogramming Partial Bank C...\r\n");
        TMR_PPC_mWriteSlaveReg1(0x42000000, 0x10);
        program_icap((Xuint32 *)0x20423E74,74236,&HwIcap);
        assert_reset(1);
        delay(2000000);
        TMR_PPC_mWriteSlaveReg0(baseaddr, 0);
    }
    else {
        banner();
        xil_printf("No Errors Found In This Iteration
Check!\r\n\r\n");
    }
}

```

**Figure 6-6 – Infinite loop error checking code**

### 6.2.3 EDK Hardware Design

The primary goal of the EDK hardware design is to create a custom wrapper for the PowerPC that is used for decoding the logic and minority voter outputs from the replicated TMR modules and alert the PowerPC which module needs to be reprogrammed. When creating a custom wrapper for the PowerPC, a VHDL template file is automatically generated by EDK. The user must enter any additional generics and ports into the top-level wrapper file, name <wrapper\_name>.vhd. For this case, the file is called opb\_tmr.vhd. The user-defined generics and ports must be added in both the entity declaration of the top-level module as well as the instantiation of the user-logic module, along with the necessary port mappings.

For this wrapper, the NUM\_TMR integer generic value was defined. This generic value should match the corresponding value at the top-level VHDL file for the entire self-healing system. It is used to determine how many outputs must be passed through voting circuitry and therefore how many logic and minority voter signals must be compared by the wrapper. The user-defined ports for the custom PowerPC wrapper are shown in Figure 6-7. The tmr2ppc\_logic and tmr2ppc\_min\_voters inputs receive the logic and minority voter outputs from the partial region, respectively. For each set of voting circuitry instantiated in the design, three corresponding sets of logic and minority voter outputs must be decoded by the PowerPC, so the busses are of width  $3 * \text{NUM\_TMR}$ . The ppc2tmr\_rst and ppc2tmr\_bus\_macro\_en outputs control the resetting and enabling of the three partially reconfigurable regions. For both 3-bit vectors, bit 0, the most significant bit (MSB), is used to control the first partial module, bit 1 is used to control the second partial module, and bit 2, the least significant bit (LSB), is used to control the third partial module. Please note that the ppc2tmr\_bus\_macro\_en output is not actually used to control the bus macro enable pins, which was its original intention. Instead, it controls custom enable functionality implemented in the design because errors were encountered when trying to use enable inputs to the bus macros.

Since the PowerPC uses the Big Endian bit naming convention, in an N-bit bus, bit 0 is the most significant bit and bit N-1 is the least significant bit [13]. Therefore, the “to” notation for standard logic vectors must be used in the PowerPC wrapper design instead of the more common “downto” notation. In fact, all standard logic vectors in the entire design, including the

custom VHDL modules, make use of Big Endian notation. This makes communication with the PowerPC easier.

```
tmr2ppc_logic      : in  std_logic_vector(0 to 3*(NUM_TMR)-1);
tmr2ppc_min_voters : in  std_logic_vector(0 to 3*(NUM_TMR)-1);
ppc2tmr_rst       : out std_logic_vector(0 to 2);
ppc2tmr_bus_macro_en : out std_logic_vector(0 to 2);
```

**Figure 6-7 – User-Defined Ports for the Custom PowerPC Wrapper**

After the top-level wrapper file is modified, the `user_logic.vhd` file must be modified to implement the custom logic for the wrapper. Once again, the user-defined ports and generics must be added to the entity description. The error decoding process used in the custom wrapper is shown in Figure 6-8. This process is based on an 80 Hz clock signal generated by a `clk_convert` module. As discussed in the Error Sampling Analysis Section 6.1.4, this was tested to be a good clock speed for checking for errors. A faster clock may read different values from the three partially reconfigurable modules based on slight differences in propagation delay rather than actual errors in the circuitry.

On the rising edge of the 80 Hz clock signal when the reset signal is not enabled, the process checks if slave register 0 has a value of 0. This register is used to store the value of the module that needs to be reprogrammed by the PowerPC. If it has a value of 0, then no modules are currently in error. If a module is in error (the register is non-zero), the error decoding process will not check for any more errors. Once the PowerPC has reconfigured a partial module, it must set the register value to 0 so that this error decoding process can resume its error checking functionality.

During the error-decoding process, a for-loop iterates `NUM_TMR` times to loop through all of the logic and minority voter values. The input logic and minority voter signals to the wrapper are formatted in such a way that all of the values corresponding to the first partial module take up the first `NUM_TMR` bits, all of the values corresponding to the second partial module take up the second `NUM_TMR` bits, and all of the values corresponding to the third and final partial module take up the third `NUM_TMR` bits. Therefore, two temporary 3-bit vector are used to concatenate the logic and minority voter values for the value corresponding to the current iteration. Once these values are determined, the output logic values are checked using a case-statement for any value that is in the minority. If such a case is found, the corresponding

partial module that needs to be reconfigured is written to slave register 0. If no such case is found (i.e. all of the partial modules output the same logic value), then the minority voters are checked using a second embedded case-statement to see if any majority voter is outputting a different value than the rest of the majority voters, as determined by the minority voter outputs. If so, the corresponding partial module that needs to be reconfigured is written to slave register 0.

```

error_reg : process (slow_clk)
    variable logic_temp      : std_logic_vector(0 to 2);
    variable min_voters_temp : std_logic_vector(0 to 2);
begin
    if slow_clk'event and slow_clk = '1' then
        if Bus2IP_Reset = '1' then
            slv_reg0_temp <= (others => '0');
            logic_temp    := (others => '0');
            min_voters_temp := (others => '0');
        else
            if (slv_reg0 = x"00000000") then
                for i in 0 to NUM_TMR-1 loop
                    logic_temp := tmr2ppc_logic(i) & tmr2ppc_logic(i + NUM_TMR)
                                & tmr2ppc_logic(i + 2*NUM_TMR);
                    min_voters_temp := tmr2ppc_min_voters(i)
                                    & tmr2ppc_min_voters(i + NUM_TMR)
                                    & tmr2ppc_min_voters(i + 2*NUM_TMR);
                    case logic_temp(0 to 2) is
                        when "001" => slv_reg0_temp <= x"00000003";
                        when "010" => slv_reg0_temp <= x"00000002";
                        when "100" => slv_reg0_temp <= x"00000001";
                        when "110" => slv_reg0_temp <= x"00000003";
                        when "101" => slv_reg0_temp <= x"00000002";
                        when "011" => slv_reg0_temp <= x"00000001";
                        when others =>
                            case min_voters_temp(0 to 2) is
                                when "001" => slv_reg0_temp <= x"00000003";
                                when "010" => slv_reg0_temp <= x"00000002";
                                when "100" => slv_reg0_temp <= x"00000001";
                                when others =>
                                    end case;
                            end case;
                        end loop;
                    else
                        slv_reg0_temp <= (others => '0');
                    end if;
                end if;
            end if;
        end process error_reg;

```

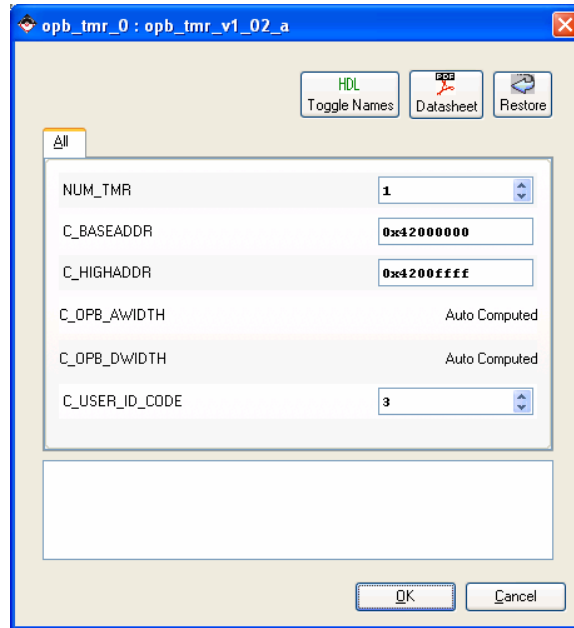
**Figure 6-8 – Error Decoding Process for the Custom PowerPC Wrapper**

While slave register 0 is used for alerting the PowerPC which partially reconfigurable module is in error, slave register 1 is used for the PowerPC to control the reset and bus macro enable signals corresponding to the three partial modules. The least significant three bits of the 32-bit register are used to control the reset functionality. The next bit is not used, and then the following three bits are used to control the enable functionality. The fourth bit is skipped to more easily allow the PowerPC to set these values using 4-bit hexadecimal values. The mapping of these signals is shown in Figure 6-9 below.

```
ppc2tmr_rst          <= slv_reg1(29 to 31);  
ppc2tmr_bus_macro_en <= slv_reg1(25 to 27);
```

**Figure 6-9 – Reset and Enable Mapping for the Custom PowerPC Wrapper**

Once the custom PowerPC wrapper has been implemented, it must be imported back into the EDK project. The step-by-step process for creating and importing a custom wrapper are detailed in the Production Flow in Section 6.3, specifically steps (4) through (7). Once the wrapper has been re-imported into the EDK project, it must be connected to the desired bus and a valid unique address range must be set. To modify the NUM\_TMR generic to enable error decoding of additional voted outputs, right-click on the module in the “System Assembly View” in EDK and select “Configure IP...” For this project, the name of the module to select is `opb_tmr_v1_02_a`. A configuration window as shown in Figure 2-1 below will include the user-definable value for the NUM\_TMR generic. Once this value is changed, the “Hardware > Generate Netlist” process must be run to create the updated netlist files.



**Figure 6-10 – Configuration Window for the opb\_tmr Custom PowerPC Wrapper in EDK**

Once all of the netlist files have been synthesized in EDK, one final modification to the EDK hardware design must be made externally to the project. The DCM module must be removed from its wrapper and placed at the top-level system.vhd file of the EDK project. The step-by-step process for this modification is given in the Production Flow in Section 6.3, specifically steps (9) through (11). This process requires the modification of the top-level system.vhd file to directly instantiate the DCM module instead of the DCM wrapper. Also, the synthesis settings file for the top-level system must include a reference to the location of the library containing the DCM module. A command-line synthesis must then be executed to update the netlist for the top-level system module. It is important to note, however, that this changed will not update the configuration of the actual EDK project. Therefore, it is advised that a backup copy of the modified system VHDL and synthesis parameter files be kept externally from the EDK project since they may be overwritten by EDK. Also, if any changes must be made to the EDK project and the “Generate Netlist” command is re-executed, the modified system files will be overwritten and will have to be re-generated externally.

## 6.2.4 Logic Module

The logic unit designed for this self-healing system is an LED-blinking circuit that blinks an output LED at the rate of 1 Hz. When an “error” occurs to the module, the output LED blinks at 8 Hz instead of 1 Hz. This is caused by the error\_in\_n signal, which is attached to a push-button on the development board in the user constraints file. Under usual operation, the module uses a counter to count from 0 to 50,000,000, after which the output is inverted and the counter is reset to 0. Since the Virtex-II Pro FPGA has a 100 MHz clock, this results in an output where the LED is on for 0.5 seconds and off for 0.5 seconds, which is a 1 Hz signal. After an error occurs, though, the counter only counts up to 6,250,000, which results in an 8 Hz output LED blinking signal.

Since this module must be synchronized with the other two replicated modules, especially after a reconfiguration, additional logic for the majority-voted feedback signal must be implemented. Due to the internal counters of the modules, the majority feedback signal cannot simply be used as the only source of next-state logic. For example, if two modules are currently outputting 0 with a count of 25,000,000 and the newly reprogrammed module is also outputting 0 but with a count of 10,000,000, the state machines will appear to be synchronized, but the two majority modules will transition to the 1 state before the reprogrammed module. The counter values therefore must also be reset when the feedback signal changes. Figure 6-11 below shows the code used to synchronize the states based on the feedback signal. The module checks if the feedback signal has switched its value from either 0 to 1 or 1 to 0. This is accomplished by registering the feedback signal, thereby delaying it by a clock cycle, and comparing that value to the current feedback value. When these two values are not equal, the feedback has switched states. The two counter values are then reset to 0 and the two blinking signals are set to the current feedback value. This resynchronizes the module to the majority output of the other blinking modules.

```
if (feedback_in_1d /= feedback_in) then
    counter_slow := 0;
    counter_fast := 0;

    blink_slow_signal <= feedback_in;
    blink_fast_signal <= feedback_in;
end if;
```

**Figure 6-11 – Feedback Logic for the Blinking LED Module**

## 6.2.5 Partial Module

As described in the analysis section of this chapter, each partial module includes a logic module, a majority voter, and a minority voter. Therefore, the top-level partial module containing these modules must include inputs for the logic and majority voter outputs from the other two reconfigurable modules. It must also contain any inputs and outputs necessary for the logic module, which is just the majority-voted feedback signal in this case. Each partial module must output its logic module, majority voter, minority voter, and final vote output values. The final voter output is defined in Figure 6-12 below. This final voter outputs the majority voter signal when it is not in the minority; otherwise, it outputs 0. Therefore, the majority voter value is output to the final voter signal only if that majority voter value is itself not in the minority.

```
voter_out <= majority_out_signal and not minority_out_signal;
```

**Figure 6-12 – Final Voter Output for the Partial Module**

In order to make this partial module useable for any number of signals to be voted upon, the NUM\_TMR generic is used to generate the proper number of majority and minority voters. NUM\_TMR is defined at the top-level file as the number of outputs and signals in this system that must be passed through voting circuitry. The proper number of majority and minority voters are instantiated using generate statements, as shown in Figure 6-13 below. All of the related logic, majority voter, and minority voter signals are defined as busses with generic width of NUM\_TMR bits. This allows for the use of generate statements using for-loops, where each iteration through the loop read from or writes to the corresponding bit value in the vectors. Each majority voter instantiated uses an output bit from the LED blinking module along with the corresponding outputs bits from the replicated logic modules in the other partial regions. The majority output value is stored in the corresponding bit of the majority output signal. Likely, for each minority voter instantiated, one bit of the output of the majority voter along with the corresponding output majority voters bits from the other partial regions are used to determine if the majority voter in this partial module is different than the other two majority voters.



```

majority_voters : for i in 0 to NUM_TMR-1 generate
  maj_voter : majority_voter
    port map (
      A => logic_signal(i),
      B => logic_in_1_en(i),
      C => logic_in_2_en(i),
      Y => majority_out_signal(i));
end generate majority_voters;

minority_voters : for i in 0 to NUM_TMR-1 generate
  min_voter : minority_voter
    port map (
      P => majority_out_signal(i),
      R1 => maj_voter_in_1_en(i),
      R2 => maj_voter_in_2_en(i),
      Y => minority_out_signal(i));
end generate minority_voters;

```

**Figure 6-13 – Generic Majority and Minority Voter Instantiation in the Partial Module**

Finally, the enable signals from the PowerPC are also used to disable any inputs from module that are currently being reconfigured, as shown in Figure 6-14 below. Since the proper use of the bus macro enable signals could not be determined in this project, the enable signals were implemented in VHDL in each of the partially reconfigurable modules. When the active low enable signals are ‘0’, the majority and logic outputs from the corresponding partially reconfigurable module are passed to the voting circuitry in this module. Otherwise, if a module is being reconfigured, the PowerPC will set the enable lines to ‘1’, which will result in the majority and logic values being ‘0’.

```

maj_voter_in_1_en <= maj_voter_in_1 when enable_1_n='0' else (others=>'0');
maj_voter_in_2_en <= maj_voter_in_2 when enable_2_n='0' else (others=>'0');

logic_in_1_en <= logic_in_1 when enable_1_n='0' else (others=>'0');
logic_in_2_en <= logic_in_2 when enable_2_n='0' else (others=>'0');

```

**Figure 6-14 – Input Enables for the Partial Module**

Please note that the design and synthesis constraints detailed in Section 5.2.6 of this report must be followed for implementation of all of the partial modules. Specifically, these modules must not contain any clock or reset primitives. For the synthesis of these modules, the “**Keep Hierarchy**” synthesis option must be enabled and the “**Add I/O Buffers**” synthesis option must be disabled.

## 6.2.6 Safe Module

The safe module used in this VHDL design is designated as the area of the circuit that would not be affected by SEUs. In a true TMR implementation, this area would most likely be implemented on a separate PCB (Printed Circuit Board), which would be much less susceptible to errors than an FPGA. The safe module is used to OR the three final voter outputs from the partial modules, as shown in Figure 6-15 below. The logical OR-ing of these final voter outputs represents taking the majority output value of the three majority voters. Also, active low enable signals are implemented to only use the values of the voters when they are not being reconfigured. If a module is in error and is currently being reprogrammed, the PowerPC will assert a '1' for the corresponding enable signal of the module. In the safe module, the corresponding voter signal will be set to '0' so that it does not affect the output of the final OR gate used to compare all of the final voter signals.

```
voter_1_en <= voter_1 when enable_n(0) = '0' else (others => '0');
voter_2_en <= voter_2 when enable_n(1) = '0' else (others => '0');
voter_3_en <= voter_3 when enable_n(2) = '0' else (others => '0');

X <= voter_1_en or voter_2_en or voter_3_en;
```

**Figure 6-15 – Safe Region VHDL Code**

## 6.2.7 Generic Top-Level TMR

The design of the top-level module for the self-healing system must follow all of the design constraints set forth in Section 5.2.4, which describes the design of the top-level file for the self-reconfiguring system. A hierarchical design approach must again be followed, such that all static and partial modules must be instantiated in the top-level file as black boxes, with no logic at the top-level. Section 5.2.4 details some specific characteristics and synthesis parameters necessary for developing and implementing a top-level module.

The top-level module in this design includes three partially reconfigurable modules instead of the one partially reconfigurable module in the self-reconfiguring system. Each of the three modules includes a logic unit, a majority voter, and a minority voter. The static modules in

this design include the PowerPC system and the “safe region.” All of the necessary bus macros for any I/O between a partially reconfigurable module and the rest of the system are also instantiated and connected in the top-level module. As discussed in the partial module and EDK wrapper subsections, the NUM\_TMR generic value is used to determine how many copies of the voting circuitry must be instantiated and connected properly. NUM\_TMR corresponds to the number of outputs or signals that must be voted upon. The actually majority and minority voters are instantiated in the lower-level partial module, but additional bus macros must be generated at the top-level module to properly connect all of the separate voting modules. All bus macros used in this system are asynchronous, narrow bus macros without enable signals.

The bus macro modules are instantiated in using generate statements, much like the instantiation of the voting circuitry in the partial modules. The first bus macro instantiation is used to connect the non-voting inputs to each of the partially reconfigurable modules. The code for this instantiation is given in Figure 6-16 below. It uses a generate statements to create three bus macros, one for each of the three partially reconfigurable modules. All of the corresponding signals are stored as three-bit vectors to easily map to each of the partial modules. Since the logic module used in this self-healing system is a relatively simple blinking LED module, only three non-voting inputs are necessary. The majority-voted feedback signal is used for synchronizing the three modules. The identical feedback signal is used as an input for each partially reconfigurable module, but it still must be passed through three separate bus macros because each bus macro corresponds to only one partially reconfigurable module. The reset signal from the PowerPC for the corresponding module must also be passed through the bus macro to reset the module’s state after a reconfiguration. Lastly, the corresponding input error signals, used to modify the frequency of the blinking in the modules, must also be mapped to the corresponding partial module. The rest of the bus macro inputs are tied to ‘0’ and their outputs are left open. If additional logic inputs are necessary for this module, they would be declared in this portion of the code.

```

input_bus_macros_all : for i in 0 to 2 generate
inputbusleft : busmacro_xc2vp_l2r_async_narrow
port map (
input0 => X_signal(0), output0 => feedback_in_signal(i),
input1 => '0',
input2 => ppc2tmr_rst_signal(i), output2 => rst_signal(i),
input3 => error_in(i), output3 => error_in_signal(i),
input4 => '0',
input5 => '0',
input6 => '0',
input7 => '0');

```

**Figure 6-16 – Instantiation of the Input Bus Macros**

The remaining inputs for the voting circuitry are instantiated in the code shown in Figure 6-17 below. This piece of VHDL code only demonstrates the bus macros instantiation for the inputs of the voting circuitry for partial module 0. The bus macro instantiation for the other two partial modules are similar. Each partial module must have the inputs for the logic and majority outputs from the other two partial modules. In this case, for partial module 0, it must take the logic and majority signals from partial modules 1 and 2. All of the necessary logic and majority voter signals are busses of width NUM\_TMR. Also, NUM\_TMR defined the number of bus macros to instantiate to properly connect all of these signals. Each iteration through the for-loop maps the corresponding bit of the logic and majority voter signals.

```

input_bus_macros_0 : for i in 0 to NUM_TMR-1 generate
inputbusleft_0 : busmacro_xc2vp_l2r_async_narrow
port map (
input0 => logic_1_signal(i), output0 => logic_in_1_signal_0(i),
input1 => logic_2_signal(i), output1 => logic_in_2_signal_0(i),
input2 => majority_1_signal(i), output2 => majority_in_1_signal_0(i),
input3 => majority_2_signal(i), output3 => majority_in_2_signal_0(i),
input4 => '0',
input5 => '0',
input6 => '0',
input7 => '0');

```

**Figure 6-17 – Instantiation of the Voter Input Bus Macros for Partial Module 0**

The logic and voting outputs of each partially reconfigurable module must likewise be mapped to bus macros, as shown in Figure 6-18 below. This piece of VHDL code only demonstrates the bus macro instantiation for the outputs of partial module 0. The outputs for the other two partial modules are instantiated in a similar manner. Much like the voter input bus macros, the number of output bus macros instantiated for a single partially reconfigurable module is defined by the NUM\_TMR generic. Each bus macro module is used to map a single bit of the logic, majority voter, minority voter, and final voter signals to the corresponding vectors of width NUM\_TMR.

```

output_bus_macros_0 : for i in 0 to NUM_TMR-1 generate
  outputbusright_0 : busmacro_xc2vp_l2r_async_narrow
  port map (
    input0 => logic_out_0_signal(i), output0 => logic_0_signal(i),
    input1 => majority_out_0_signal(i), output1 => majority_0_signal(i),
    input2 => minority_out_0_signal(i), output2 => minority_0_signal(i),
    input3 => voter_out_0_signal(i), output3 => voter_0_signal(i),
    input4 => '0',
    input5 => '0',
    input6 => '0',
    input7 => '0');
end generate;

```

**Figure 6-18 – Instantiation of the Logic and Voter Output Bus Macros for Partial Module 0**

The final set of bus macros instantiated in the top-level module are for the enable signals from the PowerPC. Each partially reconfigurable module must read the enable signals from the other two partial modules to determine which values should be used in the voting circuitry. If a module is currently being reconfigured, the PowerPC will disable the corresponding enable signal, and the value should not be used in the voting circuitry of the other two modules. The bus macro used for the enable signals for partial module 0 is shown in Figure 6-19 below. The bus macro instantiation for the enable signals of the other two partial modules follows the same pattern.

```

inputbusleft2_0 : busmacro_xc2vp_l2r_async_narrow
port map (
  input0 => ppc2tmr_bus_macro_en_signal(1), output0 => enable_1_n_signal_0,
  input1 => ppc2tmr_bus_macro_en_signal(2), output1 => enable_2_n_signal_0,
  input2 => '0',
  input3 => '0',
  input4 => '0',
  input5 => '0',
  input6 => '0',
  input7 => '0');

```

**Figure 6-19 – Instantiation of the Enable Bus Macro for Partial Module 0**

Lastly, the logic and minority voter signals from each of the three partial modules must be passed to the PowerPC system for error decoding. The VHDL code for this is shown in Figure 6-20 below. The logic signals from the three partial modules are concatenated into a bus of width 3\*NUM\_TMR. In this bus, the first NUM\_TMR bits are for the output logic signal(s) from partial module 0, the second NUM\_TMR bits are for the output logic signal(s) from partial module 1, and the final NUM\_TMR bits are for the output logic signals from partial module 2. This is also true for the minority voter signals.

```

tmr2ppc_logic_signal      <= logic_0_signal & logic_1_signal & logic_2_signal;
tmr2ppc_min_voters_signal <= minority_0_signal & minority_1_signal & minority_2_signal;

```

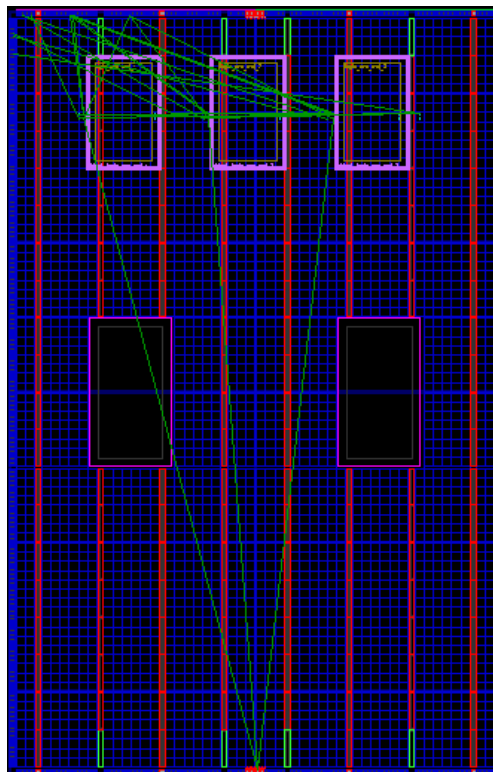
**Figure 6-20 – Concatenation of the Logic and Minority Voter Signals for the PowerPC**

After all of the necessary partial modules, static modules, and bus macros have been instantiated and connected properly, the top-level module must be synthesized. As explained in the top-level module design section of the self-reconfiguring chapter, the “**Keep Hierarchy**” and “**Add I/O Buffers**” synthesis options must be enabled for this top-level module.

## 6.2.8 PlanAhead Implementation

Once the VHDL and PowerPC design has been completed and synthesized in Xilinx ISE and EDK, several netlist files are created. The next step in the design process is to import these netlist files into PlanAhead, as described in 5.2.7: PlanAhead Implementation in the chapter

about a self-reconfiguring system. Please refer to that section for some implementation specifics regarding the PlanAhead tool being used with partially reconfigurable modules. When using PlanAhead with a self-healing system, three partially reconfigurable modules must be placed on the floorplan rather than just the single partial module in the self-reconfiguring system. The steps for drawing a PBlock and settings its attribute to a reconfigurable module must now be replicated in order to create the three unique partially reconfigurable modules. A sample floorplan for a self-healing system with three partially reconfigurable modules is shown in Figure 6-21 below. The three pink rectangles at the top of the floorplan represent the three partially reconfigurable modules. The green lines represent wiring connecting the bus macros to the rest of the board.



**Figure 6-21 – PlanAhead Floorplan for a Self-Healing System**

When using the generic template for a self-healing system, the bus macro placement is more complex due to the number of bus macros that need to be generated, even for a relatively simple system. When using the system described in this section for floorplanning, the bus macros named `input_bus_macros_all` must be placed corresponding to their module number.

Please note that all module numbers are indexed at base 0 (i.e. the three modules are numbered 0, 1, and 2). Thus, partial module 0 must have `input_bus_macros_all[0].inputbusleft` placed on the left side of the partial module, straddling the boundary between the partially reconfigurable region and the rest of the board. Likewise, `input_bus_macros_all[1].inputbusleft` and `input_bus_macros_all[2].inputbusleft` must be placed on the left side of partial modules 1 and 2, respectively. The input bus macros handling the inputs for the voting circuitry have the form `input_bus_macros_x[i].inputbusleft_x`, where  $x$  defines the partial module and  $i$  defines the bit of the NUM\_TMR-sized bus. Thus, all bus macros with the same value of  $x$  must be placed on the left side of the corresponding partially reconfigurable region. Each partially reconfigurable region must have bus macros placed for  $i$  from 0 to NUM\_TMR-1. The output bus macros follow the same format with the name `output_bus_macros_x[i].outputbusright_x`. The output bus macros must be placed on the right side of the reconfigurable area, though. Finally, the bus macros handling the enable signals are named using the convention `inputbusleft2_x`, where  $x$  again defines the partially reconfigurable area.

Once all of the static modules, partial modules, and bus macros have been placed, the floorplanning portion of the design is complete. The remainder of the design is described in the Production Flow in Section 6.3. The partial reconfiguration script files must be generated and modified as described in the Production Flow. Once the modified script files are executed, the bit files will be generated and placed in the “merge” directory of the PlanAhead floorplan folder. The `static_full.bit` file is the bitstream for the complete design, including all three partial regions. The `<partial_module>_cv_routed_partial.bit` file is the partial bitstream for the partial module whose name is given by `<partial_module>`. For ease of use, it is recommended that the number of the partially reconfigurable area is appended to its name. For instance, in this design the partial bitstreams are named `pblock_logic_and_voters_i_cv_routed_partial.bit`, where  $i$  is the number of the partial region. Likewise, `<partial_module>_blank.bit` is the blanking bitstream that will clear the corresponding partial module from the FPGA. The blanking bitstreams are not used in this self-healing system.



## 6.3 Production Flow

The following tool flow is best used with the exact versions of Xilinx tools described in Section 2.6 – Tools. This tool flow is not guaranteed with any older or newer version of these Xilinx tools. This following tool flow will provide specific instructions on how to fully integrate PowerPC functionality with a partially reconfigurable TMR design to create a self-healing system.

- 1) Open EDK and create a new project with the project wizard. Specify your specific board settings. Add desirable peripherals. Set instructions to be stored in `iocm_cntlr` and `Data/Stack/Heap` to `docm_cntlr`.
- 2) Once the project has been created, go into the IP Catalog in EDK. Under FPGA Reconfiguration, double-click on `opb_hwicap`, this will add the IP core to the system assembly view. Now connect ‘SOPB’ under `opb_hwicap_0` to the OPB bus.
- 3) Double-click on `opb_hwicap_0` and assign a 64k address space currently not in use. For example: `0x40200000` to `0x4020FFFF`.
- 4) Perform “Hardware->Create or Import Peripheral”. Click next. Select “Create templates for a new peripheral”. Click next. Select “To an XPS project” and click next. Create a name for the peripheral and click next. Go through the wizard and select the properties you want for your wrapper.
- 5) In the `pcores` directory of your EDK project folder, go into the directory of your peripheral wrapper. Go into the `hdl` directory. Go into the `vhdl` directory. Redesign the `user_logic.vhd`. In the `<module name>.vhd` file add user-defined generics and ports. Also in this file update the revision number for the module in multiple areas of the `vhdl` file.
- 6) Go back to EDK and perform “Hardware->Create or Import Peripheral”. Click next. Click on import existing peripheral. Click next. Select “To an XPS project” and click next. Select the original name and change to the new revision number you chose to define in the `vhdl` file. Click next. Select only `hdl` sources. Click next. Use existing Peripheral Analysis Order file and browse to the original wrapper directory, enter the data folder and select the `<module name>.pao` file listed. Click next. Find the custom

vhdl files in the wrapper library and remove them. Click on Add Files and add your modified wrapper vhd files. Click next. Make sure to place the peripheral on the correct bus and complete the wizard.

- 7) Add your custom wrapper to your project. Connect the peripheral to the bus and set the address range.
- 8) Perform “Software->Clean Libraries”, then perform “Software->Generate libraries and BSPs”, lastly perform “Device Configuration->Update Bitstream”. Depending on your computer, this process should roughly take 15 to 20 minutes.
- 9) The DCM module must be removed from the wrapper file and instantiated in the top-level EDK system.vhd file. To do this, open the system.vhd file in the hdl subfolder of the EDK project. In system.vhd, find the dcm\_0\_wrapper component declaration in system.vhd. Delete the dcm\_0\_wrapper declaration and associated block\_box attribute, and instead copy and paste the dcm\_module declaration, which can be found in the dcm\_0\_wrapper.vhd file. Also, copy the four attributes from the entity of the dcm\_0\_wrapper.vhd file to the entity of the system.vhd file, changing the entity name from dcm\_0\_wrapper to system where necessary. Next, change the dcm\_0\_wrapper instantiation for instance dcm\_0 to dcm\_module. The port mapping remains the same, but the generic map must be copied from the instantiation in the dcm\_0\_wrapper.vhd file. Finally, add the following library declaration to the top of the system.vhd file:

```
library dcm_module_v1_00_a;  
use dcm_module_v1_00_a.All;
```

- 10) Now, the modified system.vhd file must be synthesized outside of EDK. It is recommended that the modified system.vhd file is copied to another location for backup purposes since EDK may overwrite it during a netlist generation. Open the synthesis subfolder in the EDK project directory. Edit the dcm\_0\_wrapper\_xst.prj file, and copy the first line that defines where the dcm\_module library is located. Next, Edit the system\_xst.prj synthesis file and paste the library location as the first line of the file. It is also recommended that a backup copy of the system\_xst.prj file is kept outside of the EDK folder because it may also be overwritten.

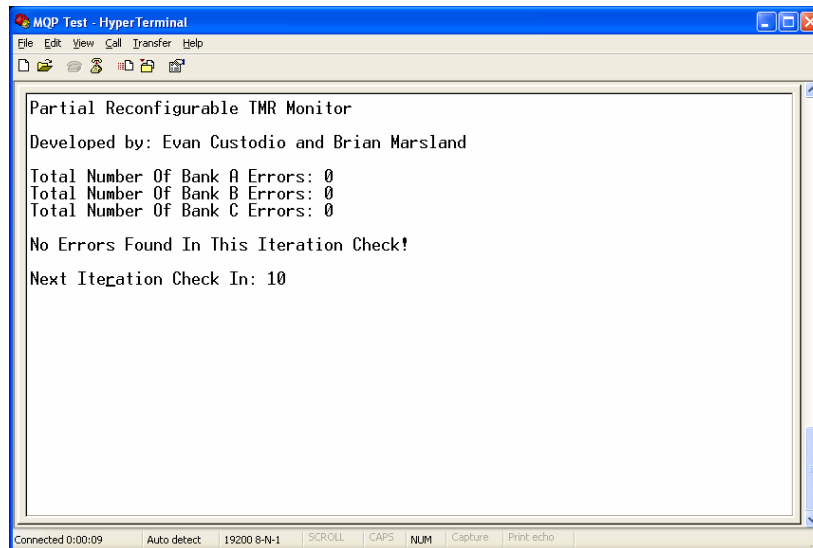
- 11) Open a DOS shell, and change the directory to the synthesis subfolder of the EDK project directory. Run the command “**xst -ifn system\_xst.scr**” to synthesize the new system.ngc file, which can be found in the implementation folder. Please note that this updated system.ngc file will be overwritten by EDK the next time the “Generate Netlist” process is launched.
- 12) Open ISE and create a new project with your board settings. Create a top level VHDL module. This module will include port mapping references to any static or partial designs and can be based off of the TMR top level design.
- 13) Keep in mind, the static and partial modules to be linked into the top level should all comply with the synthesis design rules of the partial reconfiguration flow described in “Dynamic Partial Reconfiguration of a Field Programmable Gate Array”. Make sure to synthesize each individual project.
- 14) In your top-level ISE project add an existing source (not a copy). Import the \*.XMP file created in your EDK project into ISE.
- 15) Map all ports at the top level so that the EDK project, the static modules and the partial modules share all port constraints with no conflicts.
- 16) Locate the UCF file of the XPS project and add it as the top level UCF of the ISE project.
- 17) Add any new port mappings to the UCF for the other static/partial files if needed. Now double-click on implementation and stop the process once it processes and creates a new BMM file.
- 18) Open PlanAhead and create a new project. Import the top-level NGC file created during synthesis. Point reference to any folders which contain NGC files for child modules described within the top-level NGC. Select the correct part number for your board, import the UCF file created in ISE and click finish.
- 19) Perform “File->Update Netlist...”. Click on the “Replace a specific module” radio button. Browse for the “system.ngc” file in the EDK implementation directory. Also, point reference to the EDK implementation directory and click next. Replace the original “system” with the new one you are loading and click next.
- 20) For every static module, right click on the static module and click create new pblock.

- 21) Manually input the follow command into PlanAhead's command line:  
*HDI::param set -name project.enablePR -bvalue yes*
- 22) For every partial module, right click on the partial module and click on draw a pblock.
- 23) Size out the modules and place the bus macros according to the partial reconfiguration flow described in "Dynamic Partial Reconfiguration of a Field Programmable Gate Array".
- 24) For every partial module on the floorplan, add the MODE attribute to it and set it to RECONFIG.
- 25) Next perform "File->Export Floorplan". In the window, set the option to partial reconfig and click on finish.
- 26) Next perform "Tools-> Run Partial Reconfig", In the window click on generate script files only and click on finish.
- 27) Go to the floorplan plan directory created and place the BMM file created from the ISE top-level project into the floorplan directory. Open the static folder and edit the "staticlogicImpl.bat" file. Next, append the "-bm ..\bmmfilename.bmm" flag to the ngdbuild command.
- 28) Go back to the floorplan directory and execute BuildAll.bat.
- 29) After the script is complete, the merge folder should contain your full bit file, your partial bit file and your blank partial bit file.
- 30) Before programming the bit file to the board, ensure that the Mode Pins are set to Slave Serial Mode (M0 = 1, M1 = 1. M2 = 2). The Boundary Scan Mode disables the ICAP. JTAG configuration is still available, though, because it overrides other configuration modes [11].

For considerations for PowerPC software development in EDK and the storage of the partial bit files in memory, please refer to Sections 5.3.1 and 5.3.2, respectively, from the production flow of a self-reconfiguring system.

## 6.4 Testing and Results

Much like the testing of the self-reconfiguring system, the proof-of-concept self-healing system uses the LED display and the RS-232 interface for visual testing purposes. An oscilloscope is also used to observe the LED outputs more closely, especially during the reconfiguration of a partial module. The self-healing system designed in this project should be capable of displaying an output 1 Hz blinking LED even if a module is in error. Also, the PowerPC will poll the outputs approximately every ten seconds, and will then reconfigure any modules in error as necessary. An “error” can be placed on any of the three replicated logic modules through a corresponding button push. The module will then output an 8 Hz blinking LED. A successful self-healing system will allow for any module to go into error without affecting the final voted output or the other static regions. The system must also be able to recover from a single error by reprogramming and re-synchronizing the module in error.



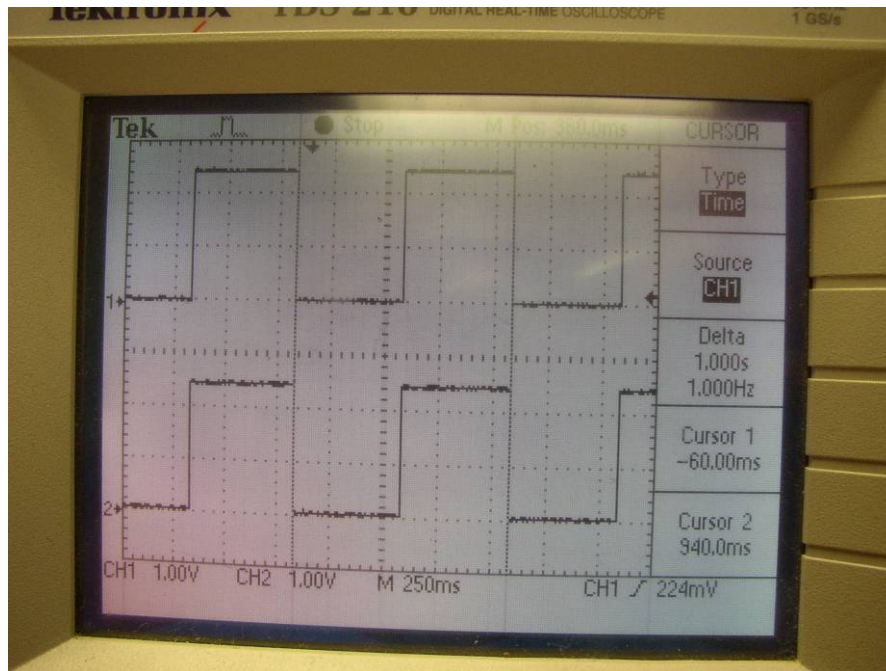
```
MQP Test - HyperTerminal
File Edit View Call Transfer Help
Partial Reconfigurable TMR Monitor
Developed by: Evan Custodio and Brian Marsland
Total Number Of Bank A Errors: 0
Total Number Of Bank B Errors: 0
Total Number Of Bank C Errors: 0
No Errors Found In This Iteration Check!
Next Iteration Check In: 10
Connected 0:00:09 Auto detect 19200 8-N-1 SCROLL CAPS NUM Capture Print echo
```

Figure 6-22 – HyperTerminal Output for Initial Operation with No Errors

When the full bitstream is programmed to the board over the JTAG interface, the system works as expected. The top three LEDs display the non-voted logic output values of the three replicated logic modules. The bottom three LEDs display the minority voter outputs for the corresponding partial modules. The bottom-right LED displays the final output of the entire TMR circuitry. Before inflicting an error on any of the modules, the three logic outputs and the final voter output all display a synchronized 1 Hz blinking signal. The three minority voters all

output a 0 (the LED is on due to inverted logic) since all of the majority voters are outputting the same value. The HyperTerminal window displaying the status messages sent through the RS-232 interface is shown in Figure 6-22 above.

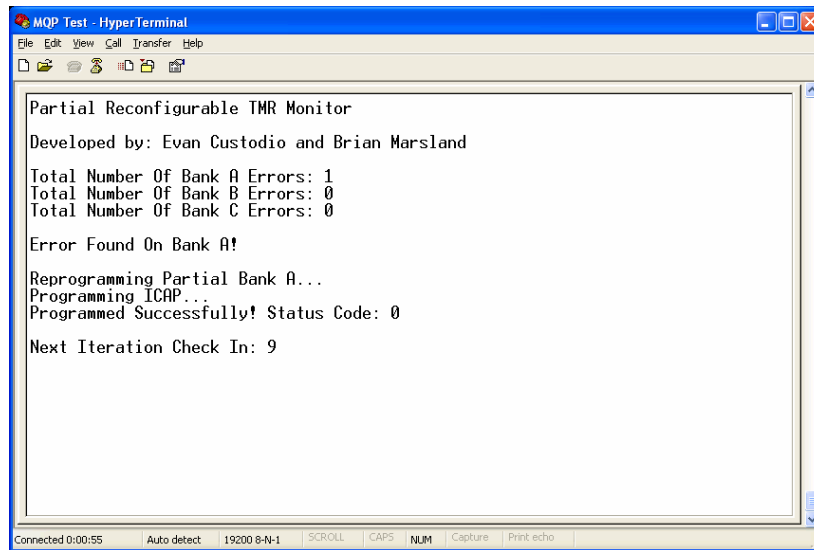
The corresponding oscilloscope reading for the self-healing system without a module in error is shown in Figure 6-23 below. The top signal displays the logic output for the first module and the bottom signal displays the final voted output. The two signals are synchronized at 1 Hz. Although it is not shown, the output logic signals from all modules were verified to be synchronized at 1 Hz.



**Figure 6-23 – Oscilloscope Reading of Synchronized 1 Hz Logic and Final Voter Signals**

When a button is pushed to cause an error on the first module the corresponding logic output for that module is changed to the 8 Hz blinking signal without affecting the other two modules or the final output. Also, in this case, all of the minority voters still output a 0 since the three majority voter modules still output the same values, even if one module is wrong. Since no logic was implemented to cause an error on the majority voters, the minority voters should always output a 0. After a variable time less than ten seconds, the module in error is reprogrammed to a 1 Hz LED blinking output and is synchronized with the other two modules. There is a variable time for reprogramming since the PowerPC uses a slow ten second polling cycle to check for errors. In a real system, the error would be detected and the module would be

reprogrammed much more quickly to lessen the chance of an error on a second module while the first module is still in error. The HyperTerminal window on the PC displays a status message indicating that an error on the first partial module (Bank A) has been detected and that the module is being reprogrammed. This status window also keeps count of how many times each module is being reprogrammed. The HyperTerminal window for an error on the first module is shown in Figure 6-24 below.



**Figure 6-24 – HyperTerminal Output for an Error on Bank A**

In order to verify the operation of the final voted output while one module is in error, an oscilloscope is used to more accurately monitor the levels of the signals, as shown in Figure 6-25 below. The top signal in the oscilloscope reading is the output of the first module, which is in error. The bottom signal is the final voted output. Once the error occurs, the top signal switches to an 8 Hz signals while the final output remains at 1 Hz. There are no glitches on the final output while the first module is in error, even during the reprogramming of the first module. There are glitches on the 8 Hz signal, though, when the majority signal switches states. This is based on the feedback logic implemented in VHDL and is not representative of a problem with a self-system. After the first module is reprogrammed, it is reset to the 0 state (please note that the LEDs used inverted logic, so the oscilloscope values are inverted). Then, once the majority-voted feedback signal switches states, the reprogrammed module switches states too and is then synchronized with the other two modules.

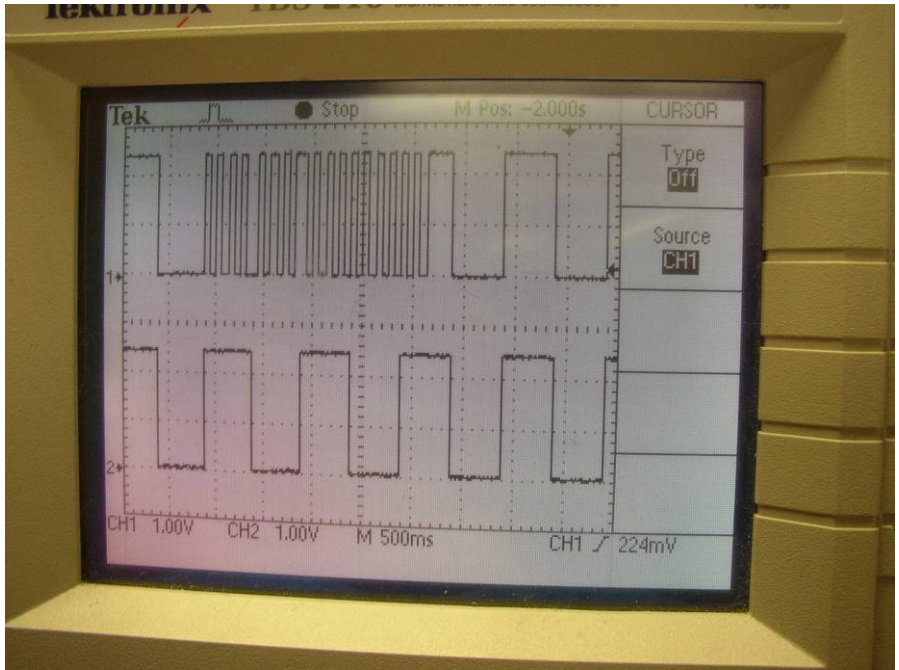


Figure 6-25 – Oscilloscope Reading of One Module in Error with Valid Final Output

The resynchronization of the reconfigured module with the majority output is shown more closely in Figure 6-26 below. The output of the reconfigured module is a 1 Hz signal in sync with the other two partial modules.

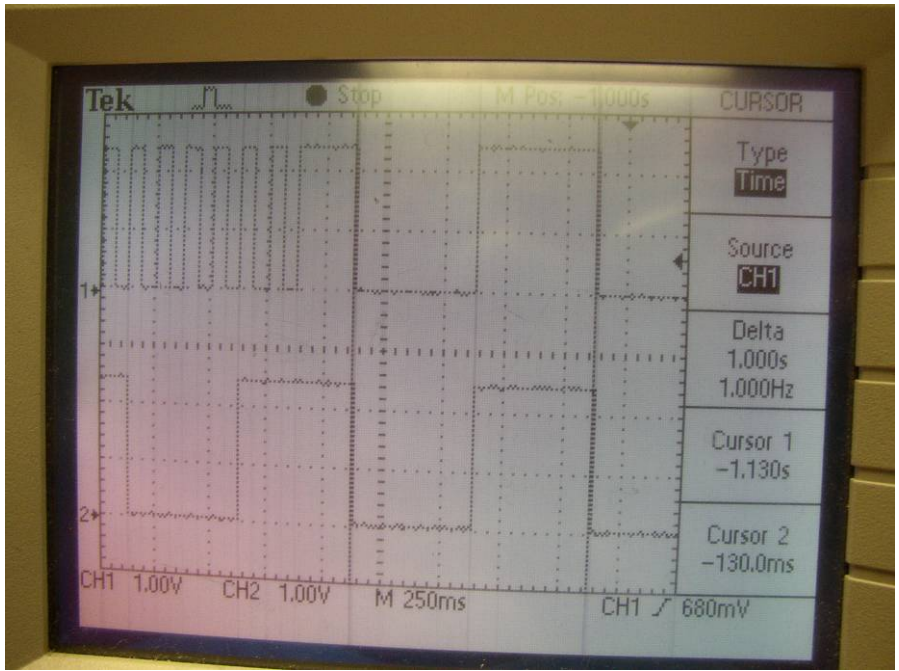
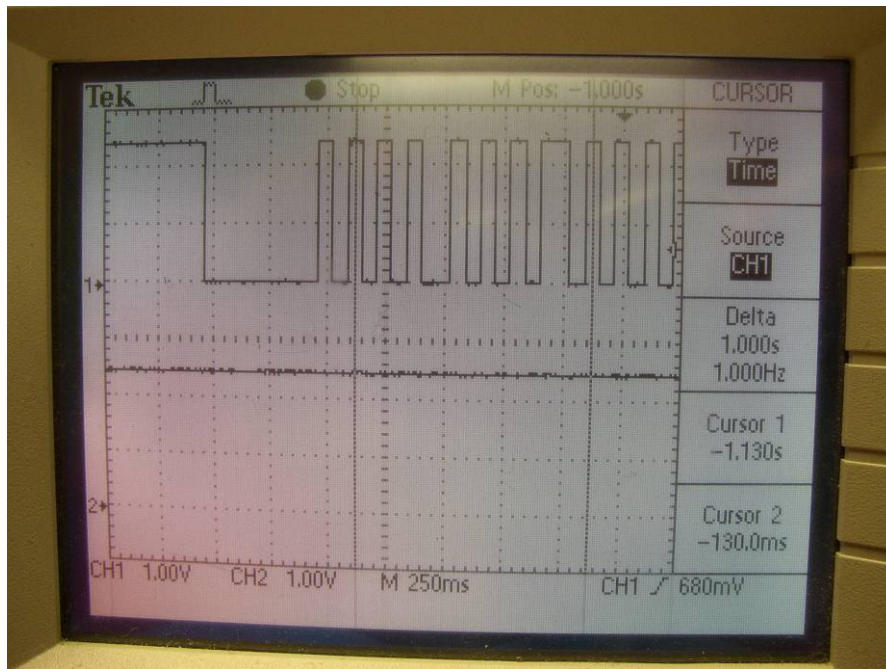


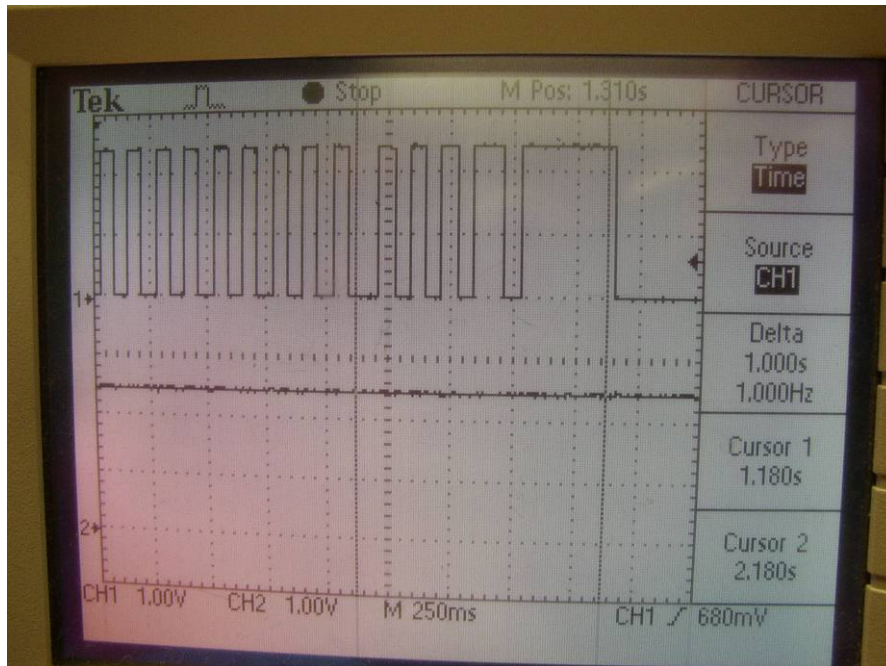
Figure 6-26 – Oscilloscope Reading of a Reconfigured Module being Resynchronized



The final oscilloscope readings are used to verify the logic and minority voter outputs of the partial module while it is in error. This is also used to determine if there are any glitches in the output during reconfiguration. Figure 6-27 shows the logic and minority voter outputs of the module as it enters into error, and Figure 6-28 shows the same outputs as the module is reprogrammed. While the module is in error, the value of the minority voter is 0. This is because the minority voter determines if one of the majority voters differs from the others. Although the logic output of the first module is wrong, the corresponding majority voter should still output the same 1 Hz signal. Even during reprogramming, though, the minority output of the module being reconfigured does not appear to have any glitches or unexpected values. Although logic has been implemented to only read this signal when the enable signal is asserted by the PowerPC, it does appear that the output does not glitch during reprogramming. It is still recommended, however, that the enable signals are used for accurate and expected circuit operation.



**Figure 6-27 – Oscilloscope Reading of the Outputs of a Module in Error**



**Figure 6-28 – Oscilloscope Reading of the Outputs of a Module during Reconfiguration**

Finally, errors were inflicted on the remaining two logic modules in order to fully test the self-healing TMR implementation. As shown in Figure 6-29 and Figure 6-30, these errors are correctly detected and reprogrammed. Thus, errors are properly detected in all three modules and are properly reprogrammed and resynchronized as necessary. The final voted output of this self-healing system remains valid if any one module is in error.

```

MQP Test - HyperTerminal
File Edit View Call Transfer Help
Partial Reconfigurable TMR Monitor
Developed by: Evan Custodio and Brian Marsland
Total Number Of Bank A Errors: 1
Total Number Of Bank B Errors: 1
Total Number Of Bank C Errors: 0
Error Found On Bank B!
Reprogramming Partial Bank B...
Programming ICAP...
Programmed Successfully! Status Code: 0
Next Iteration Check In: 5
Connected 0:01:28 Auto detect 19200 8-N-1 SCROLL ICAPS NUM Capture Print echo

```

**Figure 6-29 – HyperTerminal Output for an Error on Bank B**

```
Partial Reconfigurable TMR Monitor
Developed by: Evan Custodio and Brian Marsland
Total Number Of Bank A Errors: 1
Total Number Of Bank B Errors: 1
Total Number Of Bank C Errors: 1
Error Found On Bank C!
Reprogramming Partial Bank C...
Programming ICAP...
Programmed Successfully! Status Code: 0
Next Iteration Check In: 5
```

**Figure 6-30 – HyperTerminal Output for an Error on Bank C**

Behavior of this self-healing system was also observed for multiple concurrent errors. This system is actually able to recover from two non-simultaneous errors. If two buttons are pushed sequentially, both modules will blink at 8 Hz. This 8 Hz signal then becomes the majority signal, and is therefore fed back to the third module, which is not in error. Due to the feedback logic, however, this third module actually outputs an 8 Hz signal. Since the error signal is latched in the PowerPC wrapper until the PowerPC reads it every ten seconds, the first module that had an error will get reprogrammed first. Then, the final output will revert to the correct 1 Hz mode since that is now the majority. The wrapper will again check for errors, and the second module that had an error will get reprogrammed. All three modules are now the correct, synchronized 1 Hz modules. This behavior is not likely to occur in a real system, however, since the PowerPC will not have the lengthy delay between reprogramming errors. This delay was implemented for the purpose of visually displaying the self-healing system. In a real system, it would check for errors much more often. Also, due to the low probability of SEUs and SETs, it is unlikely for two such errors to occur so close together. It is interesting to note, though, that a system may be able to recover from two errors close together, even if the output is invalid for a short time. If the errors had occurred within the same clock cycle of the error decoding process, it is unknown what the majority would be and the system would therefore be unpredictable. The case of two concurrent errors can be recovered from only if the error from the first signal is latched before the second signal has an error.

In the case of inflicting three concurrent errors on this test system, the three modules will become synchronized 8 Hz modules, and the final output will be the 8 Hz blinking signal. Since this is now the majority value, the modules will not get reprogrammed since they will not appear to be in error. In a real system, however, it is not likely that three errors would cause the same result in all three modules, so the system behavior would become unpredictable.

The proof-of-concept system implemented in this portion of the project passed all of the desired tests that demonstrate the functionality of a self-healing system. An error on any one module does not affect the final output, and the module with the error is properly reprogrammed and resynchronized with the other modules. The rest of the circuitry remains in valid operation while one module is in error or is being reprogrammed. Thus, the self-healing system is capable of detecting an error on a single module and gracefully recovering from the error without any affect on the final output value.

## 7 Future Considerations

Although the TMR self-healing design protects the modules to a much higher degree, there is still a possibility of certain portions of the circuitry being affected by radiation and compromising the entire system. These areas will be discussed in much further detail and are points of future research in this area. The current weaknesses in the design rely on a few unprotected areas of the FPGA. The Bus Macros and basic internal routing between partial modules and other modules can still fall into error. The PowerPC wrappers that bridge onboard system components to the FPGA's PowerPC core are unprotected. The partial bitstreams which flow through the ICAP for partial reconfiguration may be affected during reconfiguration. The software storage of the partial bitstreams and the PowerPC software need to be placed on a secure storage medium. Lastly, there are further considerations to take when implementing TMR on more complex state machines.

### 7.1 Bus Macros and Internal Routing

The bus macros as mentioned before are the main bridging between partial modules and other modules around it. These macros contain logic created by Xilinx to ensure proper routing to partial modules before and after reconfiguration. The issue with bus macros is that they are no different than any other logic module on the FPGA. Therefore, these bus macros are susceptible to the SEUs which could affect its functionality. At the moment, it is unknown how to partially reconfigure these bus macros, or if it is even possible. Having any of these bus macros malfunction due to the SEUs will cripple the entire self-healing system.

Similar to the bus macros, there exists internal routing outside of the partial regions which connect bus macros to PowerPC wrappers. This routing is also unprotected from the radiation. Future research can be done to guarantee that these routings do not fail. If any of these routings were to fail, the communication between the custom peripheral and the TMR system will sever. If the communication to the PowerPC were to fail, then the PowerPC would not know to reconfigure any of the partial modules if they were to be placed in error.

## 7.2 PowerPC Wrappers

The PowerPC wrappers are somewhat of an abstraction between the PowerPC and the FPGA, but they are vital logic modules that interface the PowerPC with any other hardware. These logic modules are not seen on the PlanAhead floorplan, but they do get placed and routed. Similar to the bus macros, these logic modules are unprotected on the FPGA. It is conceivable that these modules could be partially reconfigured into their own partial areas, but there is large overhead complexity of bus macro routing that is involved for this to occur. For future research, it could be true that the PowerPC architecture may be too large of an overhead to act as a simple partial reconfiguration controller. Therefore, looking into a smaller VHDL implementation or PicoBlaze-based solutions may provide a simpler overhead that can be partially reconfigured. Also, the choice of removing the partial reconfiguration controller off the FPGA could solve this problem and would provide a smaller probability of error.

## 7.3 ICAP

The ICAP as previously mentioned before, is a physical interface port which provides in-circuit reconfiguration for the FPGA. On Virtex-II Pro devices, this physical port is located on the lower right-hand corner of the FPGA. [8] This interface reads configurations off of a neighboring block ram which contains partial bitstream specifications for reconfiguring portions of the FPGA. The main concern with this part of the self-healing system is the susceptibility of radiation onto the block ram buffer which contains the partial bitstream specifications. If the bitstream data were to be damaged by radiation, the ICAP interface could be reprogramming and damaging areas on the FPGA instead of healing them. As mentioned in the previous section, if the partial reconfiguration controller was taken off the FPGA onto a custom chip, the controller could be hardwired directly to either the ICAP or JTAG interface for partial configuration thus reducing the probability of error on the FPGA.

## 7.4 Software Storage

There are two main areas of software storage on the current self-healing design. The PowerPC execution code is located on portion of block ram on the FPGA. The partial bitstreams which feed into the ICAP for reconfiguration are location on a static non-volatile flash storage. These two storages areas are unprotected from radiation and corruption may occur. If the partial streams on flash were to be modified, these bitstreams could damage the FPGA instead of healing it. It is recommended that all software should also be redundant and compared on a periodic basis. That way if any software faced corruption from radiation, it could also heal from a redundant copy elsewhere on the system.

## 7.5 TMR Considerations

One limitation to the TMR system is when two modules are placed into error before any one of the modules can heal. When this unique situation occurs, it causes voting circuitry to view the modules in error as a majority. This undesirable event should be looked into further. One possible solution could be to make the TMR system itself triple redundant. This does not alleviate the possibility of two modules going into error at the same time, but it reduces the probability of that event compromising the system. Instead of two modules having to go into error at the same time, four modules on two TMR systems would have to go it error at the same time in order to affect the output. If logic overhead is not a concern in the application, this would be a possible solution.

Another consideration one must face in the TMR system design is the points of convergence and divergence. The point where inputs are split before reaching the redundant modules must occur outside of the FPGA. The reason for this is because any radiation which affects the FPGA can sever these routings. Each input and output to and from the TMR system must have its own I/O point in and out of the FPGA. Similarly, the point where the redundant outputs are placed through a logical OR must occur outside of the FPGA. This logical OR operation is the most critical area of the TMR system and must be handled as a highly sensitive circuit. It is recommended that this be placed off board and well protected from radiation.

## 7.6 State Machine Considerations

It is natural that any partial module that is being reconfigured may lose state synchronization with the other two redundant modules. Further research must look into not just healing the circuitry of the state machine, but healing and resynchronizing the state machine information with the other two partial modules. For simple state machines, this problem is solved by forcing the next state of the module outside of the module, through TMR with the other next state values and back into the module itself. When doing this, the designer must be careful that the error sampling rate is slow enough so that the newly reconfigured state machine will have ample time to synchronize before error checking. In more complex state machine designs, there could be states in the state machine that do not get updated with next state before error checking. States may get initialized at some point after reset, and may never get updated by next state. These states could still affect the output without getting synchronized and can place the entire partial area into infinite error loop. It is recommended that a faster resynchronization method be implemented in order to avoid this dangerous loop.



## 8 Conclusion

The design and implementation of a self-healing system was completed successfully in this project. All of the goals set forth at the beginning of the project were satisfactorily met. The complete development environment and production flow used to achieve the final self-healing system are fully documented to allow future designs to implement a similar error detection and correction system. The final design also included considerations for future expandability and reusability to assist in any future implementations based off of the findings in this project.

The first main goal of this project was to implement a self-reconfiguring system. Although the project team's efforts from the previous term yielded a partially reconfigurable system, they were unable to reconfigure the device without the need for an external PC. Thus, before implementing any TMR circuitry for the self-healing system, a solution for self-reconfiguration had to be discovered. Without the ability of an FPGA to reconfigure portions of its own circuitry, a true self-healing system would not be possible. Instead, the design would have to resort to scrubbing and reprogramming the entire board. This would affect the outputs of the system, which is undesirable. This was not an issue, though, as the self-reconfiguring system was successfully designed.

Following the completion of the self-reconfiguring system, the next main goal of the project was to implement a simple proof-of-concept TMR-based self-healing system. First, the VHDL modules for the majority and minority voting circuits were created. After careful consideration regarding the formation of the partial safe regions, it was decided that the system would use three partial modules, each containing a logic module, a majority voter, and a minority voter. This simple proof-of-concept system was successfully implemented using a 4-input XOR module. Specific details for this design were not included in this report since the following design better demonstrates the TMR-based self-healing system.

The final portion of this project involved the design of a more complex self-reconfiguring system. Although there was not enough time in the project duration to create a complex algorithmic module as originally planned, a smaller system was designed to better demonstrate the functionality of the self-healing system. This system was a counter-based LED blinking circuit. Since it is a sequential circuit, it also makes use of the majority-voted feedback signal to successfully resynchronize any reconfigure modules. Thus, this example circuit better

demonstrates the functionality of the self-healing system than the previous XOR module. Also, the TMR voting logic for this system was redesigned to generically instantiate the correct voting circuitry and interconnects, which assists in the reusability of the code in this project.

Therefore, all of the goals of this project have been successfully achieved. The final result of a complete TMR-based self-healing system passed all tests for the verification of its functionality. The process used to create this system has been described in detail for any potential future applications using this concept. Also, several future design considerations and suggestions were made regarding concepts outside of the scope of this project. These considerations should be reviewed before implementing a self-healing system as an actual part of a real system. Overall, the research and implementation details of a self-healing system discovered in the project may potentially assist in a more secure design of an FPGA-based mission-critical system.

## References

- [1] "Avnet," Avnet, Inc., 2007, <http://www.em.avnet.com/>.
- [2] Blodget, Brandon, Philip James-Roxby, Eric Keller, Scott McMillan, Prasanna Sundararajan, "A Self-Reconfiguring Platform," Proceedings of the 13th International Conference on Field Programmable Logic and Applications, Lisbon, Portugal, September 1-3, 2003, pp.565-574.
- [3] Brown, Stephen D., Robert J. Francis, Jonathan Rose, Zvonko G. Vranesic, Field-Programmable Gate Arrays, Springer, 1992.
- [4] Carmichael, Carl, "Xilinx Application Note 197 (v1.0.1): Triple Module Redundancy Design Techniques for Virtex FPGAs," Xilinx, July 2006, <http://www.xilinx.com/bvdocs/appnotes/xapp197.pdf>.
- [5] Disabello, Douglas Michael, "Fault Tolerant FPGA Co-Processing Toolkit," Boston University, Boston, MA, 2006.
- [6] Dorairaj, Nij, Eric Shiflet, Mark Goosman, "PlanAhead Software as a Platform for Partial Reconfiguration," Xcell Journal, vol. 2005, no. 55, pp. 68-71, 2005, [http://www.xilinx.com/publications/xcellonline/xcell\\_55/xc\\_prmethod55.htm](http://www.xilinx.com/publications/xcellonline/xcell_55/xc_prmethod55.htm).
- [7] "Early Access Partial Reconfiguration User Guide," Xilinx PR Early Access Lounge, <http://www.xilinx.com/support/prealounge/protected/docs/ug208.pdf>, March 2006.
- [8] Eck, Vince, Punit Kalra, Rick LeBlanc, Jim McManus, "Xilinx Application Note 662 (v2.4): In-Circuit Partial Reconfiguration of RocketIO Attributes," Xilinx, May 2004, <http://www.xilinx.com/bvdocs/appnotes/xapp662.pdf>.
- [9] Kao, Cindy, "Benefits of Partial Reconfiguration," Xcell Journal, vol. 2005, no. 55, pp. 65-67, 2005, [http://www.xilinx.com/publications/xcellonline/xcell\\_55/xc\\_reconfig55.htm](http://www.xilinx.com/publications/xcellonline/xcell_55/xc_reconfig55.htm).
- [10] Kristan, Michael, Brian Loveland, Robert Sazanowicz, "Dynamic Partial Reconfiguration of a Field Programmable Gate Array," Worcester Polytechnic Institute, Worcester, MA, 2007.
- [11] Lagger, Arnaud, "Self-Reconfigurable Platform for Cryptographic Application," School of Computer and Communication Sciences, Swiss Federal Institute of Technology Lausanne, February 2006, [http://rdsg.epfl.ch/webdav/site/rdsg/users/128366/public/master\\_project/report.pdf](http://rdsg.epfl.ch/webdav/site/rdsg/users/128366/public/master_project/report.pdf).
- [12] Lundy, Michael P., "A Self-Healing Circuit Implementing TMR," Worcester Polytechnic Institute, Worcester, MA, 2006.
- [13] "PowerPC Processor Reference Guide," Xilinx, September 2003, [http://direct.xilinx.com/bvdocs/userguides/ppc\\_ref\\_guide.pdf](http://direct.xilinx.com/bvdocs/userguides/ppc_ref_guide.pdf).

- [14] Rose, Jonathan, Abbas El Gamal, Alberto Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays," Proceedings of the IEEE, vol. 81, no. 7, pp. 1013-1029, July 1993.
- [15] "TMR Tool User Guide UG156 v1.0," Xilinx, September 2004, <http://www.xilinx.com/products/milaero/ug156.pdf>.
- [16] "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet DS083 v4.6," Xilinx, March 2007, <http://direct.xilinx.com/bvdocs/publications/ds083.pdf>.
- [17] "WinHex: Computer Forensics & Data Recovery Software, Hex Editor & Disk Editor," X-Ways Software Technology AG, 2007, <http://www.x-ways.net/winhex/index-m.html>.
- [18] "Xilinx Application Note 290 (v1.2): Two Flows for Partial Reconfiguration: Module Based or Difference Based," Xilinx, September 2004, <http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf>.
- [19] "Xilinx TMRTool: The First Triple Module Redundancy Development Tool for Reconfigurable FPGAs," Xilinx, 2006, [http://www.xilinx.com/esp/mil\\_aero/collateral/tmrtool\\_sellsheet\\_wr.pdf](http://www.xilinx.com/esp/mil_aero/collateral/tmrtool_sellsheet_wr.pdf).
- [20] Zahiri, Behrooz, "Structured ASICs: Opportunities and Challenges," Proceedings of the 21st International Conference on Computer Design, IEEE, pp. 404-409, 2003.
- [21] Zeineddini, Amir H. Sheikh, "Secure Partial Reconfiguration of FPGAs," George Mason University, Fairfax, VA, Summer 2005, [http://ece.gmu.edu/courses/Crypto\\_resources/web\\_resources/theses/GMU\\_theses/Zeineddini/Zeineddini\\_Summer\\_2005.pdf](http://ece.gmu.edu/courses/Crypto_resources/web_resources/theses/GMU_theses/Zeineddini/Zeineddini_Summer_2005.pdf).

# Appendix A: Self-Reconfiguring System

## A.1: PowerPC Software

```
/*
 *
 * reprogram.c - Xilinx hardware ICAP programming tool
 *
 * WPI/General Dynamics Electrical & Computer Engineering MQP D2007
 * Self-Healing Partial Reconfiguration of an FPGA
 * By Evan Custodio and Brian Marsland
 *
 * This file contains implementation code that will assist the
 * parent application with programming a Xilinx FPGA. This code
 * is designed to work with the on-board PowerPC processor on
 * a Virtex-II Pro FPGA.
 *
 * This source code uses event polling of the buttons on board to
 * trigger self-reconfiguration of the FPGA by using the ICAP
 */
*****

#include <xparameters.h>
#include <xhwicap.h>
#include <xhwicap_clb_lut.h>
#include <stdio.h>
#include "xgpio.h"
#include "xstatus.h"

#define HWICAP_DEVICEID          XPAR_OPB_HWICAP_0_DEVICE_ID
#define XHI_TARGET_DEVICEID     XHI_XC2VP30 /*XHI_READ_DEVICEID_FROM_ICAP*/
#define TEST_COL                16        /* Test Column for LUT */
#define TEST_ROW                144       /* Test Row for LUT */
#define LUT_SIZE                16        /* The number of bits in a LUT */
#define NUM_READS               10       /* How many times to read back */
#undef PAUSE                     /* Pause after each readback */
#undef HWICAP_PRINT_ALL_DATA     /* Print all data */
#define MAX_COUNT               0xFFFF   /* LUT hold 16-bit values */
#define Print_xil_printf        /* A smaller footprint printf */

XStatus HwIcapLutExample(Xuint16 DeviceId);
static XHwIcap HwIcap;
Xuint8 LutWriteBuffer[LUT_SIZE];        /* Value written to the LUT */
Xuint8 LutReadBuffer[LUT_SIZE];        /* Value read back from the LUT */

int main(void)
{
    XStatus Status;
    Print("Test...\r\n");
    Status = HwIcapLutExample(HWICAP_DEVICEID);

    if (Status != XST_SUCCESS)
    {
        return XST_FAILURE;
    }

    return XST_SUCCESS;
}

XStatus HwIcapLutExample(Xuint16 DeviceId)
{
    XStatus Status;        /* Return value */
    XGpio GpioInput;      /* The driver instance for GPIO Device configured as I/P */
```

```

#ifdef PAUSE
    Xuint8 Ch;                /* For reading from UART */
#endif /* PAUSE */

Status = XHwIcap_Initialize(&HwIcap, DeviceId, XHI_TARGET_DEVICEID);
if (Status == XST_DEVICE_IS_STARTED)
{
    Print("Device is already initialized.");
}
else if (Status != XST_SUCCESS)
{
    Print("Failed to initialize: %d\r\n", Status);
    return XST_FAILURE;
}
Print("ICAP initialized...\r\n");
Status = XGpio_Initialize(&GpioInput, XPAR_PUSH_BUTTONS_3BIT_DEVICE_ID);
if (Status != XST_SUCCESS)
{
    return XST_FAILURE;
}
XGpio_SetDataDirection(&GpioInput, 1, 0xFFFFFFFF);

Xuint32 DataRead;

int button_pressed = 0;
int i;

while (1) {
    DataRead = XGpio_DiscreteRead(&GpioInput, 1);
    if (button_pressed == 0) {
        if (DataRead == 0x3) {
            Print("Programming the blank file... \r\n");
            program_icap((Xuint32 *)0x04300000, 54264, &HwIcap);
            button_pressed = 1;
        }
        if (DataRead == 0x5) {
            Print("Programming the partial file... \r\n");
            program_icap((Xuint32 *)0x0430D3F8, 54264, &HwIcap);
            button_pressed = 1;
        }
    }
    if (DataRead == 0x7) {
        for (i = 0; i < 3000000; i++);
        if (DataRead == 0x7) {
            button_pressed = 0;
        }
    }
}

int program_icap(Xuint32 * bit_file_addr, int size, XHwIcap * HwIcap) {
    Print("programming ICAP...\r\n");

    XStatus Status = XHwIcap_SetConfiguration(HwIcap, bit_file_addr, (size/4));

    if (Status != XST_SUCCESS)
    {
        Print("Failed to program: %d\r\n", Status);
        return XST_FAILURE;
    }
    XHwIcap_CommandDesync(HwIcap);
    Print("Programmed successfully: %d\r\n", Status);

    return 0;
}

```

## A.2: Top-Level – VHDL

```
-----  
-----  
-- Worcester Polytechnic Institute  
-- General Dynamics C4 Systems MQP  
-- Evan Custodio  
-- Brian Marsland  
-- D Term 2007  
-- Self-Healing Partial Reconfiguration of an FPGA  
-- Filename: top.vhd  
-----  
-----  
-- This module is the top-level module declaration for this self-reconfiguring  
-- design. It instantiates all partial and static modules under it and  
-- interconnects them safely with proper use of bus macros.  
-----  
-----  
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.STD_LOGIC_ARITH.all;  
use IEEE.STD_LOGIC_UNSIGNED.all;  
use busmacro_xc2vp_pkg.all;  
  
entity top is  
  port (  
    fpga_0_RS232_2_RX_pin          : in    std_logic;  
    fpga_0_RS232_2_TX_pin          : out   std_logic;  
    fpga_0_Push_Buttons_3Bit_GPIO_in_pin : in    std_logic_vector(0 to 2);  
    fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin  : inout std_logic_vector(0 to 31);  
    fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin : out   std_logic_vector(0 to 11);  
    fpga_0_SDRAM_8Mx32_1_SDRAM_DQM_pin : out   std_logic_vector(0 to 3);  
    fpga_0_SDRAM_8Mx32_1_SDRAM_WEn_pin  : out   std_logic;  
    fpga_0_SDRAM_8Mx32_1_SDRAM_CKE_pin  : out   std_logic;  
    fpga_0_SDRAM_8Mx32_1_SDRAM_CS_n_pin : out   std_logic;  
    fpga_0_SDRAM_8Mx32_1_SDRAM_CAS_n_pin : out   std_logic;  
    fpga_0_SDRAM_8Mx32_1_SDRAM_RAS_n_pin : out   std_logic;  
    fpga_0_SDRAM_8Mx32_1_SDRAM_Clk_pin   : out   std_logic;  
    fpga_0_SDRAM_8Mx32_1_SDRAM_BankAddr_pin : out   std_logic_vector(0 to 1);  
    fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin  : inout std_logic_vector(0 to 31);  
    fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin : out   std_logic_vector(0 to 11);  
    fpga_0_SDRAM_8Mx32_2_SDRAM_DQM_pin : out   std_logic_vector(0 to 3);  
    fpga_0_SDRAM_8Mx32_2_SDRAM_WEn_pin  : out   std_logic;  
    fpga_0_SDRAM_8Mx32_2_SDRAM_CKE_pin  : out   std_logic;  
    fpga_0_SDRAM_8Mx32_2_SDRAM_CS_n_pin : out   std_logic;  
    fpga_0_SDRAM_8Mx32_2_SDRAM_CAS_n_pin : out   std_logic;  
    fpga_0_SDRAM_8Mx32_2_SDRAM_RAS_n_pin : out   std_logic;  
    fpga_0_SDRAM_8Mx32_2_SDRAM_Clk_pin   : out   std_logic;  
    fpga_0_SDRAM_8Mx32_2_SDRAM_BankAddr_pin : out   std_logic_vector(0 to 1);  
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin : out   std_logic_vector(9 to 29);  
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin : inout std_logic_vector(0 to 31);  
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin : out   std_logic_vector(0 to 3);  
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_WEN_pin : out   std_logic;  
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_OEN_pin : out   std_logic_vector(0 to 1);  
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_CEN_pin : out   std_logic_vector(0 to 1);  
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_RPN_pin : out   std_logic;  
    fpga_0_SysACE_CompactFlash_SysACE_CLK_pin : in    std_logic;  
    fpga_0_SysACE_CompactFlash_SysACE_MPA_pin : out   std_logic_vector(6 downto 0);  
    fpga_0_SysACE_CompactFlash_SysACE_MPD_pin : inout std_logic_vector(15 downto 0);  
    fpga_0_SysACE_CompactFlash_SysACE_CEN_pin : out   std_logic;  
    fpga_0_SysACE_CompactFlash_SysACE_OEN_pin : out   std_logic;  
    fpga_0_SysACE_CompactFlash_SysACE_WEN_pin : out   std_logic;  
    fpga_0_SysACE_CompactFlash_SysACE_MPIRQ_pin : in    std_logic;  
    sys_clk_pin          : in    std_logic;  
    sys_rst_pin          : in    std_logic;  
    LED1                 : out   std_logic;  
    LED2                 : out   std_logic;  
    LED3                 : out   std_logic;
```

```

        LED4                : out    std_logic;
        OUTPUT              : out    std_logic_vector (3 downto 0);
        IN1                 : in     std_logic_vector (3 downto 0);
        IN2                 : in     std_logic_vector (3 downto 0)
    );
end top;

architecture Behavioral of top
component switch_light
    port (
        switches1 : in  std_logic_vector (3 downto 0);
        switches2 : in  std_logic_vector (3 downto 0);
        leds       : out std_logic_vector (3 downto 0));
end component;

component ppc_dis
    port (
        fpga_0_RS232_2_RX_pin          : in     std_logic;
        fpga_0_RS232_2_TX_pin          : out    std_logic;
        fpga_0_LEDs_8Bit_GPIO_d_out_pin : out    std_logic_vector(0 to 7);
        fpga_0_Push_Buttons_3Bit_GPIO_in_pin : in     std_logic_vector(0 to 2);
        fpga_0_DIP_Switches_8Bit_GPIO_in_pin : in     std_logic_vector(0 to 7);
        fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin : inout  std_logic_vector(0 to 31);
        fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin : out    std_logic_vector(0 to 11);
        fpga_0_SDRAM_8Mx32_1_SDRAM_DQM_pin : out    std_logic_vector(0 to 3);
        fpga_0_SDRAM_8Mx32_1_SDRAM_WEn_pin : out    std_logic;
        fpga_0_SDRAM_8Mx32_1_SDRAM_CKE_pin : out    std_logic;
        fpga_0_SDRAM_8Mx32_1_SDRAM_CSn_pin : out    std_logic;
        fpga_0_SDRAM_8Mx32_1_SDRAM_CASn_pin : out    std_logic;
        fpga_0_SDRAM_8Mx32_1_SDRAM_RASn_pin : out    std_logic;
        fpga_0_SDRAM_8Mx32_1_SDRAM_Clk_pin : out    std_logic;
        fpga_0_SDRAM_8Mx32_1_SDRAM_BankAddr_pin : out    std_logic_vector(0 to 1);
        fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin : inout  std_logic_vector(0 to 31);
        fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin : out    std_logic_vector(0 to 11);
        fpga_0_SDRAM_8Mx32_2_SDRAM_DQM_pin : out    std_logic_vector(0 to 3);
        fpga_0_SDRAM_8Mx32_2_SDRAM_WEn_pin : out    std_logic;
        fpga_0_SDRAM_8Mx32_2_SDRAM_CKE_pin : out    std_logic;
        fpga_0_SDRAM_8Mx32_2_SDRAM_CSn_pin : out    std_logic;
        fpga_0_SDRAM_8Mx32_2_SDRAM_CASn_pin : out    std_logic;
        fpga_0_SDRAM_8Mx32_2_SDRAM_RASn_pin : out    std_logic;
        fpga_0_SDRAM_8Mx32_2_SDRAM_Clk_pin : out    std_logic;
        fpga_0_SDRAM_8Mx32_2_SDRAM_BankAddr_pin : out    std_logic_vector(0 to 1);
        fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin : out    std_logic_vector(9 to 29);
        fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin : inout  std_logic_vector(0 to 31);
        fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin : out    std_logic_vector(0 to 3);
        fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_WEN_pin : out    std_logic;
        fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_OEN_pin : out    std_logic_vector(0 to 1);
        fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_CEN_pin : out    std_logic_vector(0 to 1);
        fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_RPN_pin : out    std_logic;
        fpga_0_SysACE_CompactFlash_SysACE_CLK_pin : in     std_logic;
        fpga_0_SysACE_CompactFlash_SysACE_MPA_pin : out    std_logic_vector(6 downto 0);
        fpga_0_SysACE_CompactFlash_SysACE_MPD_pin : inout  std_logic_vector(15 downto 0);
        fpga_0_SysACE_CompactFlash_SysACE_CEN_pin : out    std_logic;
        fpga_0_SysACE_CompactFlash_SysACE_OEN_pin : out    std_logic;
        fpga_0_SysACE_CompactFlash_SysACE_WEN_pin : out    std_logic;
        fpga_0_SysACE_CompactFlash_SysACE_MPIRQ_pin : in     std_logic;
        sys_clk_pin                    : in     std_logic;
        sys_rst_pin                    : in     std_logic;
        LED1                           : out    std_logic;
        LED2                           : out    std_logic;
        LED3                           : out    std_logic;
        LED4                           : out    std_logic
    );
end component;

signal sigin1, sigin2, sigout : std_logic_vector (3 downto 0);

begin

    reconfig_module : switch_light
        port map (

```



```

switches1 => sigin1,
switches2 => sigin2,
leds      => sigout);

base_module : ppc_dis
port map (
  fpga_0_RS232_2_RX_pin          => fpga_0_RS232_2_RX_pin,
  fpga_0_RS232_2_TX_pin          => fpga_0_RS232_2_TX_pin,
  fpga_0_LEDs_8Bit_GPIO_d_out_pin => open,
  fpga_0_Push_Buttons_3Bit_GPIO_in_pin => fpga_0_Push_Buttons_3Bit_GPIO_in_pin,
  fpga_0_DIP_Switches_8Bit_GPIO_in_pin => "00000000",
  fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin   => fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin,
  fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin  => fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin,
  fpga_0_SDRAM_8Mx32_1_SDRAM_DQM_pin  => fpga_0_SDRAM_8Mx32_1_SDRAM_DQM_pin,
  fpga_0_SDRAM_8Mx32_1_SDRAM_WEN_pin  => fpga_0_SDRAM_8Mx32_1_SDRAM_WEN_pin,
  fpga_0_SDRAM_8Mx32_1_SDRAM_CKE_pin  => fpga_0_SDRAM_8Mx32_1_SDRAM_CKE_pin,
  fpga_0_SDRAM_8Mx32_1_SDRAM_CS_n_pin => fpga_0_SDRAM_8Mx32_1_SDRAM_CS_n_pin,
  fpga_0_SDRAM_8Mx32_1_SDRAM_CAS_n_pin => fpga_0_SDRAM_8Mx32_1_SDRAM_CAS_n_pin,
  fpga_0_SDRAM_8Mx32_1_SDRAM_RAS_n_pin => fpga_0_SDRAM_8Mx32_1_SDRAM_RAS_n_pin,
  fpga_0_SDRAM_8Mx32_1_SDRAM_Clk_pin  => fpga_0_SDRAM_8Mx32_1_SDRAM_Clk_pin,
  fpga_0_SDRAM_8Mx32_1_SDRAM_BankAddr_pin => fpga_0_SDRAM_8Mx32_1_SDRAM_BankAddr_pin,
  fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin   => fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin,
  fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin  => fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin,
  fpga_0_SDRAM_8Mx32_2_SDRAM_DQM_pin  => fpga_0_SDRAM_8Mx32_2_SDRAM_DQM_pin,
  fpga_0_SDRAM_8Mx32_2_SDRAM_WEN_pin  => fpga_0_SDRAM_8Mx32_2_SDRAM_WEN_pin,
  fpga_0_SDRAM_8Mx32_2_SDRAM_CKE_pin  => fpga_0_SDRAM_8Mx32_2_SDRAM_CKE_pin,
  fpga_0_SDRAM_8Mx32_2_SDRAM_CS_n_pin => fpga_0_SDRAM_8Mx32_2_SDRAM_CS_n_pin,
  fpga_0_SDRAM_8Mx32_2_SDRAM_CAS_n_pin => fpga_0_SDRAM_8Mx32_2_SDRAM_CAS_n_pin,
  fpga_0_SDRAM_8Mx32_2_SDRAM_RAS_n_pin => fpga_0_SDRAM_8Mx32_2_SDRAM_RAS_n_pin,
  fpga_0_SDRAM_8Mx32_2_SDRAM_Clk_pin  => fpga_0_SDRAM_8Mx32_2_SDRAM_Clk_pin,
  fpga_0_SDRAM_8Mx32_2_SDRAM_BankAddr_pin => fpga_0_SDRAM_8Mx32_2_SDRAM_BankAddr_pin,
  fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin,
  fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin,
  fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin,
  fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_WEN_pin => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_WEN_pin,
  fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_OEN_pin => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_OEN_pin,
  fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_CEN_pin => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_CEN_pin,
  fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_RPN_pin => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_RPN_pin,
  fpga_0_SysACE_CompactFlash_SysACE_CLK_pin  => fpga_0_SysACE_CompactFlash_SysACE_CLK_pin,
  fpga_0_SysACE_CompactFlash_SysACE_MPA_pin  => fpga_0_SysACE_CompactFlash_SysACE_MPA_pin,
  fpga_0_SysACE_CompactFlash_SysACE_MPD_pin  => fpga_0_SysACE_CompactFlash_SysACE_MPD_pin,
  fpga_0_SysACE_CompactFlash_SysACE_CEN_pin  => fpga_0_SysACE_CompactFlash_SysACE_CEN_pin,
  fpga_0_SysACE_CompactFlash_SysACE_OEN_pin  => fpga_0_SysACE_CompactFlash_SysACE_OEN_pin,
  fpga_0_SysACE_CompactFlash_SysACE_WEN_pin  => fpga_0_SysACE_CompactFlash_SysACE_WEN_pin,
  fpga_0_SysACE_CompactFlash_SysACE_MPIRQ_pin => fpga_0_SysACE_CompactFlash_SysACE_MPIRQ_pin,
  sys_clk_pin                               => sys_clk_pin,
  sys_rst_pin                               => sys_rst_pin,
  LED1                                       => LED1,
  LED2                                       => LED2,
  LED3                                       => LED3,
  LED4                                       => LED4
);

outputbusleft : busmacro_xc2vp_r2l_async_narrow
port map(input0 => sigout(0), output0 => OUTPUT(0),
  input1 => '0',
  input2 => '0',
  input3 => '0',
  input4 => '0',
  input5 => '0',
  input6 => '0',
  input7 => '0');

inputbusleft : busmacro_xc2vp_l2r_async_narrow
port map(input0 => IN1(0), output0 => sigin1(0),
  input1 => IN1(1), output1 => sigin1(1),
  input2 => IN1(2), output2 => sigin1(2),
  input3 => IN1(3), output3 => sigin1(3),

```

```

        input4 => '0',
        input5 => '0',
        input6 => '0',
        input7 => '0');

outputbusright : busmacro_xc2vp_l2r_async_narrow
  port map(input0 => '0',
           input1 => sigout(1), output1 => OUTPUT(1),
           input2 => sigout(2), output2 => OUTPUT(2),
           input3 => sigout(3), output3 => OUTPUT(3),
           input4 => '0',
           input5 => '0',
           input6 => '0',
           input7 => '0');

inputbusright : busmacro_xc2vp_r2l_async_narrow
  port map(input0 => '0',
           input1 => '0',
           input2 => '0',
           input3 => '0',
           input4 => IN2(0), output4 => sigin2(0),
           input5 => IN2(1), output5 => sigin2(1),
           input6 => IN2(2), output6 => sigin2(2),
           input7 => IN2(3), output7 => sigin2(3));

end Behavioral;

```

### A.3: Reconfigurable Module – LED Switch – VHDL

```

-----
-- Worcester Polytechnic Institute
-- General Dynamics C4 Systems MQP
-- Evan Custodio
-- Brian Marsland
-- D Term 2007
-- Self-Healing Partial Reconfiguration of an FPGA
-- Filename: switch_light_1.vhd
-----

-- This module simply takes all inputs from onboard switches and performs a NOT
-- operation, then outputs the result to the LEDS
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity switch_light is
  Port ( switches1 : in  STD_LOGIC_VECTOR (3 downto 0);
        switches2 : in  STD_LOGIC_VECTOR (3 downto 0);
        leds       : out STD_LOGIC_VECTOR (3 downto 0));
end switch_light;

architecture Modular of switch_light is

begin
  leds <= not switches1;

end Modular;

```

## A.4: Static Module – LED Display – VHDL

```
-----  
-----  
-- Worcester Polytechnic Institute  
-- General Dynamics C4 Systems MQP  
-- Evan Custodio  
-- Brian Marsland  
-- D Term 2007  
-- Self-Healing Partial Reconfiguration of an FPGA  
-- Filename: dis_module.vhd  
-----  
-----  
-- This module uses the clock converter module to blink 2 LEDs at 2 different  
-- clock rates. 10KHz, and 1Hz  
-----  
-----  
  
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.STD_LOGIC_ARITH.all;  
use IEEE.STD_LOGIC_UNSIGNED.all;  
  
entity dis_module is  
  port (  
    CLK      : in  std_logic;  
    BLINK    : out std_logic;  
    SOLID    : out std_logic;  
    LIGHT1   : out std_logic;  
    LIGHT2   : out std_logic);  
end dis_module;  
  
architecture Modular of dis_module is  
  
  component Clk_Convrt  
    port ( Clk_in      : in  std_logic;  
          Reset       : in  std_logic;  
          Clk_1Hz, Clk_10Hz, Clk_10KHz : out std_logic  
        );  
  end component;  
  
begin  
  
  clockconverter : Clk_Convrt  
    port map (  
      Clk_in      => CLK,  
      Reset       => '0',  
      Clk_1Hz     => BLINK,  
      Clk_10KHz  => SOLID );  
  
  LIGHT1 <= '0';  
  LIGHT2 <= '0';  
  
end Modular;
```

## A.5: Static Module – Clock Converter – VHDL

```
-----  
-----  
-- Worcester Polytechnic Institute  
-- General Dynamics C4 Systems MQP  
-- Evan Custodio  
-- Brian Marsland  
-- D Term 2007  
-- Self-Healing Partial Reconfiguration of an FPGA  
-- Filename: Clk_Convrt.vhd  
-----  
-----  
-- This module takes in a 100 Mhz clock and outputs a 10KHz, 10Hz, and 1Hz  
-- clock signal. The code is borrowed and modified from the ECE 3801 Online  
-- Laboratory Resources. The ECE 3801 site is viewable at  
-- http://ece.wpi.edu/courses/ee3801/  
-----  
-----  
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.STD_LOGIC_ARITH.all;  
use IEEE.STD_LOGIC_UNSIGNED.all;  
  
entity Clk_Convrt is  
    port ( Clk_in          : in  std_logic;  
          Reset           : in  std_logic;  
          Clk_1Hz, Clk_10Hz, Clk_10KHz : out std_logic  
        );  
end Clk_Convrt;  
  
architecture Behavioral of Clk_Convrt is  
  
    signal tmp_clk_1Hz      : std_logic := '0';  
    signal tmp_clk_10Hz     : std_logic := '0';  
    signal tmp_Clk_10KHz   : std_logic := '0';  
  
begin  
    Clk_1Hz    <= tmp_clk_1Hz;  
    Clk_10Hz   <= tmp_clk_10Hz;  
    Clk_10KHz <= tmp_Clk_10KHz;  
  
    process(Reset, Clk_in)  
        variable counter_1Hz      : integer range 0 to 25_000_000;  
        variable counter_10Hz     : integer range 0 to 2_500_000;  
        variable counter_10KHz    : integer range 0 to 2_500;  
  
        begin  
            if Reset = '1' then  
                counter_1Hz      := 0;  
                counter_10Hz     := 0;  
                counter_10KHz    := 0;  
            elsif Clk_in'event and Clk_in = '1' then  
                counter_1Hz      := counter_1Hz+1;  
                counter_10Hz     := counter_10Hz+1;  
                counter_10KHz    := counter_10KHz+1;  
  
                if counter_1Hz = 25_000_000 then  
                    tmp_clk_1Hz  <= not tmp_clk_1Hz;  
                    counter_1Hz  := 0;  
                end if;  
                if counter_10Hz = 2_500_000 then  
                    tmp_clk_10Hz <= not tmp_clk_10Hz;  
                    counter_10Hz := 0;  
                end if;  
                if counter_10KHz = 2_500 then  
                    tmp_Clk_10KHz <= not tmp_Clk_10KHz;  
                end if;  
            end if;  
        end process;  
end architecture;
```

```

        counter_10KHz := 0;
    end if;
end if;
end process;

end Behavioral;

```

## A.6: User Constraints File

```

#####
## This system.ucf file is generated by Base System Builder based on the
## settings in the selected Xilinx Board Definition file. Please add other
## user constraints to this file based on customer design specifications.
#####

```

```

Net sys_clk_pin LOC=AH17;
Net sys_rst_pin LOC=G25;
Net sys_rst_pin PULLUP;
## System level constraints
# Net sys_clk_pin TNM_NET = sys_clk_pin;
# TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 10000 ps;
Net sys_rst_pin TIG;
#NET "C405RSTCORERESETREQ" TPTHURU = "RST_GRP";
#NET "C405RSTCHIPRESETREQ" TPTHURU = "RST_GRP";
#NET "C405RSTSYSRESETREQ" TPTHURU = "RST_GRP";
#TIMESPEC "TS_RST1" = FROM CPUS THRU RST_GRP TO FFS TIG;

```

```
## IO Devices constraints
```

```
#### Module RS232_2 constraints
```

```

Net fpga_0_RS232_2_RX_pin LOC=AL14;
Net fpga_0_RS232_2_TX_pin LOC=AM14;

```

```
#### Module SRAM_256Kx32_FLASH_1Mx32 constraints
```

```

Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<29> LOC=K16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<28> LOC=AD17;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<27> LOC=AE17;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<26> LOC=K14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<25> LOC=E6;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<24> LOC=AE16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<23> LOC=AE15;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<22> LOC=AE14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<21> LOC=D5;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<20> LOC=D9;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<19> LOC=E16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<18> LOC=F17;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<17> LOC=C9;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<16> LOC=D10;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<15> LOC=D12;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<14> LOC=F16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<13> LOC=F15;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<12> LOC=K15;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<11> LOC=K12;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<10> LOC=D6;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<9> LOC=K11;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<31> LOC=G10;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<30> LOC=H14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<29> LOC=H10;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<28> LOC=F13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<27> LOC=D15;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<26> LOC=D16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<25> LOC=D13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<24> LOC=D14;

```

```

Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<23> LOC=E10;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<22> LOC=G9;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<21> LOC=E13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<20> LOC=H9;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<19> LOC=C14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<18> LOC=F14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<17> LOC=E14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<16> LOC=C13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<15> LOC=K17;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<14> LOC=J16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<13> LOC=L17;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<12> LOC=J10;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<11> LOC=L16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<10> LOC=G17;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<9> LOC=G14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<8> LOC=G15;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<7> LOC=J14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<6> LOC=J15;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<5> LOC=J12;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<4> LOC=J11;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<3> LOC=G16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<2> LOC=H16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<1> LOC=E7;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<0> LOC=E9;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin<3> LOC=AF11;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin<2> LOC=AF13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin<1> LOC=AJ13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin<0> LOC=AK14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_WEN_pin LOC=AE11;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_OEN_pin<0> LOC=H13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_OEN_pin<1> LOC=C11;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_CEN_pin<0> LOC=G13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_CEN_pin<1> LOC=F9;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_RPN_pin LOC=AL11;

```

#### Module Push\_Buttons\_3Bit constraints

```

Net fpga_0_Push_Buttons_3Bit_GPIO_in_pin<0> LOC=H25;
Net fpga_0_Push_Buttons_3Bit_GPIO_in_pin<0> PULLUP;
Net fpga_0_Push_Buttons_3Bit_GPIO_in_pin<1> LOC=G26;
Net fpga_0_Push_Buttons_3Bit_GPIO_in_pin<1> PULLUP;
Net fpga_0_Push_Buttons_3Bit_GPIO_in_pin<2> LOC=H26;
Net fpga_0_Push_Buttons_3Bit_GPIO_in_pin<2> PULLUP;

```

#### Module DIP\_Switches\_8Bit constraints

```

#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<0> LOC=G18;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<0> PULLUP;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<1> LOC=H19;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<1> PULLUP;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<2> LOC=G19;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<2> PULLUP;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<3> LOC=G20;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<3> PULLUP;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<4> LOC=H21;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<4> PULLUP;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<5> LOC=G21;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<5> PULLUP;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<6> LOC=H22;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<6> PULLUP;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<7> LOC=G22;
#Net fpga_0_DIP_Switches_8Bit_GPIO_in_pin<7> PULLUP;

```

#### Module SDRAM\_8Mx32\_1 constraints

```

Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<31> LOC=AB3;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<30> LOC=V4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<29> LOC=AB4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<28> LOC=W3;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<27> LOC=AA4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<26> LOC=W4;

```

```

Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<25> LOC=AA3;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<24> LOC=Y4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<23> LOC=Y1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<22> LOC=AA1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<21> LOC=Y2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<20> LOC=AB1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<19> LOC=AA2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<18> LOC=AC1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<17> LOC=AB2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<16> LOC=AC2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<15> LOC=AE4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<14> LOC=AF4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<13> LOC=AF3;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<12> LOC=AK4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<11> LOC=AK3;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<10> LOC=AC4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<9> LOC=AC3;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<8> LOC=AD4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<7> LOC=AD2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<6> LOC=AE2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<5> LOC=AE1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<4> LOC=AG1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<3> LOC=AF2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<2> LOC=AL1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<1> LOC=AG2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<0> LOC=AL2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin<11> LOC=AD6;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin<10> LOC=W5;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin<9> LOC=V5;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin<8> LOC=AH6;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin<7> LOC=Y6;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin<6> LOC=V7;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin<5> LOC=W6;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin<4> LOC=W7;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin<3> LOC=Y7;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin<2> LOC=AA6;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin<1> LOC=AA5;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin<0> LOC=AB7;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQM_pin<3> LOC=Y3;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQM_pin<2> LOC=W2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQM_pin<1> LOC=AD3;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQM_pin<0> LOC=AD1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_WEn_pin LOC=AE5;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_CKE_pin LOC=AB6;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_CS_n_pin LOC=AB5;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_CAS_n_pin LOC=AC6;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_RAS_n_pin LOC=AH5;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_Clk_pin LOC=AC7;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_BankAddr_pin<1> LOC=V6;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_BankAddr_pin<0> LOC=AD5;

```

#### Module SDRAM\_8Mx32\_2 constraints

```

Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<31> LOC=M3;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<30> LOC=F5;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<29> LOC=N4;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<28> LOC=F4;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<27> LOC=M4;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<26> LOC=K5;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<25> LOC=L3;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<24> LOC=L4;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<23> LOC=H1;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<22> LOC=K2;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<21> LOC=J2;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<20> LOC=L2;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<19> LOC=K1;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<18> LOC=M2;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<17> LOC=L1;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<16> LOC=M1;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<15> LOC=R3;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<14> LOC=T3;

```

```

Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<13> LOC=T4;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<12> LOC=U3;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<11> LOC=U4;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<10> LOC=P4;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<9> LOC=N3;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<8> LOC=R4;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<7> LOC=N1;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<6> LOC=P1;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<5> LOC=P2;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<4> LOC=R1;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<3> LOC=R2;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<2> LOC=U2;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<1> LOC=T2;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQ_pin<0> LOC=V2;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin<11> LOC=T6;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin<10> LOC=M6;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin<9> LOC=L5;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin<8> LOC=U6;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin<7> LOC=M7;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin<6> LOC=F7;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin<5> LOC=L6;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin<4> LOC=L7;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin<3> LOC=N7;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin<2> LOC=N6;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin<1> LOC=N5;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_Addr_pin<0> LOC=R9;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQM_pin<3> LOC=K4;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQM_pin<2> LOC=H2;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQM_pin<1> LOC=P3;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_DQM_pin<0> LOC=N2;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_WEn_pin LOC=T5;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_CKE_pin LOC=P6;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_CS_n_pin LOC=P5;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_CAS_n_pin LOC=R7;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_RAS_n_pin LOC=U5;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_Clk_pin LOC=T7;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_BankAddr_pin<1> LOC=J7;
Net fpga_0_SDRAM_8Mx32_2_SDRAM_BankAddr_pin<0> LOC=R6;

```

#### Module SysACE\_CompactFlash constraints

```

Net fpga_0_SysACE_CompactFlash_SysACE_CLK_pin LOC=D17;
Net fpga_0_SysACE_CompactFlash_SysACE_MPA_pin<0> LOC=AH8;
Net fpga_0_SysACE_CompactFlash_SysACE_MPA_pin<1> LOC=AA8;
Net fpga_0_SysACE_CompactFlash_SysACE_MPA_pin<2> LOC=AB8;
Net fpga_0_SysACE_CompactFlash_SysACE_MPA_pin<3> LOC=W8;
Net fpga_0_SysACE_CompactFlash_SysACE_MPA_pin<4> LOC=U10;
Net fpga_0_SysACE_CompactFlash_SysACE_MPA_pin<5> LOC=T10;
Net fpga_0_SysACE_CompactFlash_SysACE_MPA_pin<6> LOC=T11;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<0> LOC=V8;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<1> LOC=AB9;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<2> LOC=AC9;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<3> LOC=J8;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<4> LOC=AA9;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<5> LOC=L8;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<6> LOC=M10;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<7> LOC=N9;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<8> LOC=M9;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<9> LOC=P8;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<10> LOC=N8;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<11> LOC=T8;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<12> LOC=T9;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<13> LOC=U9;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<14> LOC=U8;
Net fpga_0_SysACE_CompactFlash_SysACE_MPD_pin<15> LOC=U11;
Net fpga_0_SysACE_CompactFlash_SysACE_CEN_pin LOC=P9;
Net fpga_0_SysACE_CompactFlash_SysACE_OEN_pin LOC=AJ8;
Net fpga_0_SysACE_CompactFlash_SysACE_WEN_pin LOC=AD8;
Net fpga_0_SysACE_CompactFlash_SysACE_MPIRQ_pin LOC=R10;

```

```
NET "LED1" LOC = E31;
```



```
NET "LED2" LOC = E32;  
NET "LED3" LOC = F31;  
NET "LED4" LOC = F30;  
  
NET "OUTPUT[0]" LOC = E1;  
NET "OUTPUT[1]" LOC = E2;  
NET "OUTPUT[2]" LOC = E3;  
NET "OUTPUT[3]" LOC = E4;  
  
NET "IN1[0]" LOC = G22;  
NET "IN1[1]" LOC = H22;  
NET "IN1[2]" LOC = G21;  
NET "IN1[3]" LOC = H21;  
NET "IN2[0]" LOC = G20;  
NET "IN2[1]" LOC = G19;  
NET "IN2[2]" LOC = H19;  
NET "IN2[3]" LOC = G18;
```

# Appendix B: TMR-Based Self-Healing System

## B.1: PowerPC Software

```
/*
 * selfhealing.c - Self-healing implementation using the ICAP and
 *                 custom peripheral.
 *
 * WPI/General Dynamics Electrical & Computer Engineering MQP D2007
 * Self-Healing Partial Reconfiguration of an FPGA
 * By Evan Custodio and Brian Marsland
 *
 * This file contains implementation code that will assist the
 * parent application with programming a Xilinx FPGA. This code
 * is designed to work with the on-board PowerPC processor on
 * a Virtex-II Pro FPGA.
 *
 * This source code uses event polling of the error status flags of
 * the custom peripheral to locate errors on the FPGA and partially
 * self-reconfigure them.
 */
*****/

#include "tmr_ppc.h"
#include <xparameters.h>
#include <xhwicap.h>
#include <xhwicap_clb_lut.h>
#include <stdio.h>
#include "xstatus.h"

#define HWICAP_DEVICEID      XPAR_OPB_HWICAP_0_DEVICE_ID
#define XHI_TARGET_DEVICEID XHI_XC2VP30

static XHwIcap HwIcap;
int bankA = 0;
int bankB = 0;
int bankC = 0;

void delay(int seed) {
    int i;
    for (i = 0; i < seed; i++);
}

int program_icap(Xuint32 * bit_file_addr, int size, XHwIcap * HwIcap) {
    xil_printf("Programming ICAP...\r\n");

    XStatus Status = XHwIcap_SetConfiguration(HwIcap, bit_file_addr, (size/4));

    if (Status != XST_SUCCESS)
    {
        xil_printf("Failed To Program! Status Code: %d\r\n", Status);
        return XST_FAILURE;
    }
    XHwIcap_CommandDesync(HwIcap);
    xil_printf("Programmed Successfully! Status Code: %d\r\n\r\n", Status);

    return 0;
}

void assert_reset(int pbank) {
    TMR_PPC_mWriteSlaveReg1(0x42000000, pbank);
    delay(300);
    TMR_PPC_mWriteSlaveReg1(0x42000000, 0);
}
```

```

void banner() {
    outbyte(0x0C);
    print("Partial Reconfigurable TMR Monitor\r\n\r\nDeveloped by: Evan Custodio and Brian
Marsland\r\n\r\n");
    xil_printf("Total Number Of Bank A Errors: %d\r\n", bankA);
    xil_printf("Total Number Of Bank B Errors: %d\r\n", bankB);
    xil_printf("Total Number Of Bank C Errors: %d\r\n\r\n", bankC);
}

int main() {
    Xuint32 baseaddr = 0x42000000;
    Xuint32 Reg32Value;
    XStatus Status;
    int i;

    Status = XHwIcap_Initialize(&HwIcap, HWICAP_DEVICEID, XHI_TARGET_DEVICEID);
    if (Status == XST_DEVICE_IS_STARTED)
    {
        xil_printf("Device is already initialized.");
    }
    else if (Status != XST_SUCCESS)
    {
        xil_printf("Failed to initialize: %d\r\n", Status);
        return 1;
    }

    xil_printf("ICAP initialized...\r\n");

    xil_printf("Starting tmr_wrapper_test...\r\n");
    TMR_PPC_mWriteSlaveReg0(baseaddr, 0);
    TMR_PPC_mWriteSlaveReg1(baseaddr, 0);
    banner();

    while (1) {
        for (i = 10; i > -1; i--) {
            delay(3000000*5);
            xil_printf("Next Iteration Check In: %d ", i);
            outbyte(0x0D);
        }

        Reg32Value = TMR_PPC_mReadSlaveReg0(baseaddr);

        if (Reg32Value == 1) {
            bankA++;
            banner();
            xil_printf("Error Found On Bank A!\r\n\r\n");
            xil_printf("Reprogramming Partial Bank A...\r\n");
            TMR_PPC_mWriteSlaveReg1(0x42000000, 0x40);
            program_icap((Xuint32 *)0x20400000, 73824, &HwIcap);
            assert_reset(4);
            delay(2000000);
            TMR_PPC_mWriteSlaveReg0(baseaddr, 0);
        }
        else if (Reg32Value == 2) {
            bankB++;
            banner();
            xil_printf("Error Found On Bank B!\r\n\r\n");
            xil_printf("Reprogramming Partial Bank B...\r\n");
            TMR_PPC_mWriteSlaveReg1(0x42000000, 0x20);
            program_icap((Xuint32 *)0x20411c78, 74236, &HwIcap);
            assert_reset(2);
            delay(2000000);
            TMR_PPC_mWriteSlaveReg0(baseaddr, 0);
        }
    }
}

```

```

    }
    else if (Reg32Value == 3) {
        bankC++;
        banner();
        xil_printf("Error Found On Bank C!\r\n\r\n");
        xil_printf("Reprogramming Partial Bank C...\r\n");
        TMR_PPC_mWriteSlaveReg1(0x42000000, 0x10);
        program_icap((Xuint32 *)0x20423E74, 74236, &HwIcap);
        assert_reset(1);
        delay(2000000);
        TMR_PPC_mWriteSlaveReg0(baseaddr, 0);
    }
    else {
        banner();
        xil_printf("No Errors Found In This Iteration Check!\r\n\r\n");
    }
}
return 0;
}

```

## B.2: PowerPC Wrapper – Top-Level Wrapper – VHDL

```

-----
-- opb_tmr.vhd - entity/architecture pair
-----
-- IMPORTANT:
-- DO NOT MODIFY THIS FILE EXCEPT IN THE DESIGNATED SECTIONS.
--
-- SEARCH FOR --USER TO DETERMINE WHERE CHANGES ARE ALLOWED.
--
-- TYPICALLY, THE ONLY ACCEPTABLE CHANGES INVOLVE ADDING NEW
-- PORTS AND GENERICS THAT GET PASSED THROUGH TO THE INSTANTIATION
-- OF THE USER_LOGIC ENTITY.
-----
--
-- *****
-- ** Copyright (c) 1995-2006 Xilinx, Inc. All rights reserved. **
-- ** ** **
-- ** Xilinx, Inc. **
-- ** XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" **
-- ** AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND **
-- ** SOLUTIONS FOR XILINX DEVICES. BY PROVIDING THIS DESIGN, CODE, **
-- ** OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, **
-- ** APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION **
-- ** THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT, **
-- ** AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE **
-- ** FOR YOUR IMPLEMENTATION. XILINX EXPRESSLY DISCLAIMS ANY **
-- ** WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE **
-- ** IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR **
-- ** REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF **
-- ** INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS **
-- ** FOR A PARTICULAR PURPOSE. **
-- ** ** **
-- *****
--
-----
-- Filename:          opb_tmr.vhd
-- Version:           1.02.a
-- Description:       Top level design, instantiates IPIF and user logic.
-- Date:              Tue Apr 03 10:53:36 2007 (by Create and Import Peripheral Wizard)
-- VHDL Standard:    VHDL'93
-----
-- Naming Conventions:
-- active low signals:      "*_n"
-- clock signals:          "clk", "clk_div#", "clk_#x"
-- reset signals:          "rst", "rst_n"

```

```

-- generics:                                "C_*"
-- user defined types:                       "*_TYPE"
-- state machine next state:                 "*_ns"
-- state machine current state:              "*_cs"
-- combinatorial signals:                   "*_com"
-- pipelined or register delay signals:      "*_d#"
-- counter signals:                          "*cnt*"
-- clock enable signals:                     "*_ce"
-- internal version of output port:          "*_i"
-- device pins:                              "*_pin"
-- ports:                                    "- Names begin with Uppercase"
-- processes:                                "*_PROCESS"
-- component instantiations:                 "<ENTITY_>I_<#|FUNC>"
-----

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library proc_common_v2_00_a;
use proc_common_v2_00_a.proc_common_pkg.all;
use proc_common_v2_00_a.ipif_pkg.all;
library opb_ipif_v3_01_c;
use opb_ipif_v3_01_c.all;

library opb_tmr_v1_02_a;
use opb_tmr_v1_02_a.all;

-----
-- Entity section
-----
-- Definition of Generics:
-- C_BASEADDR                                -- User logic base address
-- C_HIGHADDR                                -- User logic high address
-- C_OPB_AWIDTH                              -- OPB address bus width
-- C_OPB_DWIDTH                              -- OPB data bus width
-- C_USER_ID_CODE                            -- User ID to place in MIR/Reset register
-- C_FAMILY                                  -- Target FPGA architecture
--
-- Definition of Ports:
-- OPB_Clk                                   -- OPB Clock
-- OPB_Rst                                    -- OPB Reset
-- Sl_DBus                                   -- Slave data bus
-- Sl_errAck                                 -- Slave error acknowledge
-- Sl_retry                                  -- Slave retry
-- Sl_toutSup                               -- Slave timeout suppress
-- Sl_xferAck                                -- Slave transfer acknowledge
-- OPB_ABus                                  -- OPB address bus
-- OPB_BE                                    -- OPB byte enable
-- OPB_DBus                                  -- OPB data bus
-- OPB_RNW                                   -- OPB read/not write
-- OPB_select                                -- OPB select
-- OPB_seqAddr                               -- OPB sequential address
-- IP2INTC_Irpt                              -- Interrupt output to processor
-----

entity opb_tmr is
  generic
  (
    -- ADD USER GENERICS BELOW THIS LINE -----
    NUM_TMR                                : integer := 1;
    -- ADD USER GENERICS ABOVE THIS LINE -----

    -- DO NOT EDIT BELOW THIS LINE -----
    -- Bus protocol parameters, do not add to or delete
    C_BASEADDR                            : std_logic_vector    := X"00000000";
    C_HIGHADDR                            : std_logic_vector    := X"0000FFFF";
    C_OPB_AWIDTH                          : integer             := 32;
    C_OPB_DWIDTH                          : integer             := 32;
    C_USER_ID_CODE                        : integer             := 3;
  )

```

```

    C_FAMILY                : string                := "virtex2p"
  -- DO NOT EDIT ABOVE THIS LINE -----
);
port
(
  -- ADD USER PORTS BELOW THIS LINE -----
  tmr2ppc_logic            : in std_logic_vector(0 to 3*(NUM_TMR)-1);
  tmr2ppc_min_voters      : in std_logic_vector(0 to 3*(NUM_TMR)-1);
  ppc2tmr_rst             : out std_logic_vector(0 to 2);
  ppc2tmr_bus_macro_en    : out std_logic_vector(0 to 2);
  -- ADD USER PORTS ABOVE THIS LINE -----

  -- DO NOT EDIT BELOW THIS LINE -----
  -- Bus protocol ports, do not add to or delete
  OPB_Clk                 : in std_logic;
  OPB_Rst                 : in std_logic;
  Sl_DBus                 : out std_logic_vector(0 to C_OPB_DWIDTH-1);
  Sl_errAck               : out std_logic;
  Sl_retry                : out std_logic;
  Sl_toutSup              : out std_logic;
  Sl_xferAck              : out std_logic;
  OPB_ABus                : in std_logic_vector(0 to C_OPB_AWIDTH-1);
  OPB_BE                  : in std_logic_vector(0 to C_OPB_DWIDTH/8-1);
  OPB_DBus                : in std_logic_vector(0 to C_OPB_DWIDTH-1);
  OPB_RNW                 : in std_logic;
  OPB_select              : in std_logic;
  OPB_seqAddr             : in std_logic;
  IP2INTC_Irpt           : out std_logic;
  -- DO NOT EDIT ABOVE THIS LINE -----
);

attribute SIGIS : string;
attribute SIGIS of OPB_Clk      : signal is "Clk";
attribute SIGIS of OPB_Rst     : signal is "Rst";
attribute SIGIS of IP2INTC_Irpt : signal is "INTR_LEVEL_HIGH";

end entity opb_tmr;

-----
-- Architecture section
-----

architecture IMP of opb_tmr is

  -----
  -- Constant: array of address range identifiers
  -----
  constant ARD_ID_ARRAY          : INTEGER_ARRAY_TYPE :=
  (
    0 => USER_00,                -- user logic S/W register address space
    1 => IPIF_RST,                -- include IPIF S/W Reset/MIR service
    2 => IPIF_INTR                -- include IPIF Interrupt service
  );

  -----
  -- Constant: array of address pairs for each address range
  -----
  constant ZERO_ADDR_PAD        : std_logic_vector(0 to 64-C_OPB_AWIDTH-1) := (others
=> '0');

  constant USER_BASEADDR       : std_logic_vector := C_BASEADDR or X"00000000";
  constant USER_HIGHADDR       : std_logic_vector := C_BASEADDR or X"000000FF";

  constant RST_BASEADDR        : std_logic_vector := C_BASEADDR or X"00000100";
  constant RST_HIGHADDR        : std_logic_vector := C_BASEADDR or X"000001FF";

  constant INTR_BASEADDR       : std_logic_vector := C_BASEADDR or X"00000200";
  constant INTR_HIGHADDR       : std_logic_vector := C_BASEADDR or X"000002FF";

  constant ARD_ADDR_RANGE_ARRAY : SLV64_ARRAY_TYPE :=
  (

```

```

        ZERO_ADDR_PAD & USER_BASEADDR,          -- user logic base address
        ZERO_ADDR_PAD & USER_HIGHADDR,         -- user logic high address
        ZERO_ADDR_PAD & RST_BASEADDR,          -- MIR/Reset register base address
        ZERO_ADDR_PAD & RST_HIGHADDR,          -- MIR/Reset register high address
        ZERO_ADDR_PAD & INTR_BASEADDR,         -- interrupt register base address
        ZERO_ADDR_PAD & INTR_HIGHADDR          -- interrupt register high address
    );

-----
-- Constant: array of data widths for each target address range
-----
constant USER_DWIDTH          : integer          := 32;

constant ARD_DWIDTH_ARRAY     : INTEGER_ARRAY_TYPE :=
(
    0 => USER_DWIDTH,          -- user logic data width
    1 => C_OPB_DWIDTH,         -- MIR/Reset register data width
    2 => C_OPB_DWIDTH          -- interrupt register data width
);

-----
-- Constant: array of desired number of chip enables for each address range
-----
constant USER_NUM_CE          : integer          := 2;

constant ARD_NUM_CE_ARRAY     : INTEGER_ARRAY_TYPE :=
(
    0 => pad_power2(USER_NUM_CE), -- user logic number of CEs
    1 => 1,                     -- MIR/Reset register - 1 CE
    2 => 16                      -- interrupt register - 16 CEs
);

-----
-- Constant: array of unique properties for each address range
-----
constant USER_INCLUDE_DEV_ISC : boolean          := true;

constant USER_INCLUDE_DEV_PENCODER : boolean      := true;

constant ARD_DEPENDENT_PROPS_ARRAY : DEPENDENT_PROPS_ARRAY_TYPE :=
(
    0 => (others => 0),          -- user logic slave space dependent properties
(none defined)
    1 => (others => 0),          -- IPIF reset/mir dependent properties (none
defined)
    2 => (
        EXCLUDE_DEV_ISC      => 1-boolean'pos(USER_INCLUDE_DEV_ISC),
        INCLUDE_DEV_PENCODER => boolean'pos(USER_INCLUDE_DEV_PENCODER),
        others => 0)
        -- IPIF interrupt dependent properties
    );

-----
-- Constant: pipeline mode
-- 1 = include OPB-In pipeline registers
-- 2 = include IP pipeline registers
-- 3 = include OPB-In and IP pipeline registers
-- 4 = include OPB-Out pipeline registers
-- 5 = include OPB-In and OPB-Out pipeline registers
-- 6 = include IP and OPB-Out pipeline registers
-- 7 = include OPB-In, IP, and OPB-Out pipeline registers
-- Note:
-- only mode 4, 5, 7 are supported for this release
-----
constant PIPELINE_MODEL       : integer          := 5;

-----
-- Constant: user core ID code
-----
constant DEV_BLK_ID           : integer          := C_USER_ID_CODE;
-----

```

```

-- Constant: enable MIR/Reset register
-----
constant DEV_MIR_ENABLE          : integer          := 1;

-----
-- Constant: array of IP interrupt mode
-- 1 = Active-high interrupt condition
-- 2 = Active-low interrupt condition
-- 3 = Active-high pulse interrupt event
-- 4 = Active-low pulse interrupt event
-- 5 = Positive-edge interrupt event
-- 6 = Negative-edge interrupt event
-----
constant USER_IP_INTR_NUM      : integer          := 1;

constant IP_INTR_MODE_ARRAY    : INTEGER_ARRAY_TYPE :=
(
  0 => 1
);

-----
-- Constant: enable device burst
-----
constant DEV_BURST_ENABLE      : integer          := 0;

-----
-- Constant: include address counter for burst transfers
-----
constant INCLUDE_ADDR_CNTR     : integer          := 0;

-----
-- Constant: include write buffer that decouples OPB and IPIC write transactions
-----
constant INCLUDE_WR_BUF        : integer          := 0;

-----
-- Constant: index for CS/CE
-----
constant USER00_CS_INDEX       : integer          := get_id_index(ARD_ID_ARRAY,
USER00);

constant USER00_CE_INDEX       : integer          :=
calc_start_ce_index(ARD_NUM_CE_ARRAY, USER00_CS_INDEX);

-----
-- IP Interconnect (IPIC) signal declarations -- do not delete
-- prefix 'i' stands for IPIF while prefix 'u' stands for user logic
-- typically user logic will be hooked up to IPIF directly via i<sig>
-- unless signal slicing and muxing are needed via u<sig>
-----
signal iBus2IP_RdCE             : std_logic_vector(0 to calc_num_ce(ARD_NUM_CE_ARRAY)-1);
signal iBus2IP_WrCE             : std_logic_vector(0 to calc_num_ce(ARD_NUM_CE_ARRAY)-1);
signal iBus2IP_Data             : std_logic_vector(0 to C_OPB_DWIDTH-1);
signal iBus2IP_BE               : std_logic_vector(0 to C_OPB_DWIDTH/8-1);
signal iIP2Bus_Data             : std_logic_vector(0 to C_OPB_DWIDTH-1) := (others =>
'0');
signal iIP2Bus_Ack              : std_logic := '0';
signal iIP2Bus_Error            : std_logic := '0';
signal iIP2Bus_Retry            : std_logic := '0';
signal iIP2Bus_ToutSup          : std_logic := '0';
signal ENABLE_POSTED_WRITE      : std_logic_vector(0 to ARD_ID_ARRAY'length-1) :=
(others => '0'); -- enable posted write behavior
signal ZERO_IP2RFIFO_Data       : std_logic_vector(0 to
ARD_DWIDTH_ARRAY(get_id_index_iboe(ARD_ID_ARRAY, IPIF_RDFIFO_DATA))-1) := (others => '0'); --
work around for XST not taking (others => '0') in port mapping
signal ZERO_WFIFO2IP_Data       : std_logic_vector(0 to
ARD_DWIDTH_ARRAY(get_id_index_iboe(ARD_ID_ARRAY, IPIF_WRFIFO_DATA))-1) := (others => '0'); --
work around for XST not taking (others => '0') in port mapping
signal iIP2Bus_IntrEvent        : std_logic_vector(0 to IP_INTR_MODE_ARRAY'length-1) :=
(others => '0');
signal iBus2IP_Clk              : std_logic;

```



```

signal iBus2IP_Reset          : std_logic;
signal uBus2IP_Data          : std_logic_vector(0 to USER_DWIDTH-1);
signal uBus2IP_BE            : std_logic_vector(0 to USER_DWIDTH/8-1);
signal uBus2IP_RdCE          : std_logic_vector(0 to USER_NUM_CE-1);
signal uBus2IP_WrCE          : std_logic_vector(0 to USER_NUM_CE-1);
signal uIP2Bus_Data          : std_logic_vector(0 to USER_DWIDTH-1);

begin

-----
-- instantiate the OPB IPIF
-----
OPB_IPIF_I : entity opb_ipif_v3_01_c.opb_ipif
generic map
(
  C_ARD_ID_ARRAY              => ARD_ID_ARRAY,
  C_ARD_ADDR_RANGE_ARRAY     => ARD_ADDR_RANGE_ARRAY,
  C_ARD_DWIDTH_ARRAY         => ARD_DWIDTH_ARRAY,
  C_ARD_NUM_CE_ARRAY         => ARD_NUM_CE_ARRAY,
  C_ARD_DEPENDENT_PROPS_ARRAY => ARD_DEPENDENT_PROPS_ARRAY,
  C_PIPELINE_MODEL           => PIPELINE_MODEL,
  C_DEV_BLK_ID               => DEV_BLK_ID,
  C_DEV_MIR_ENABLE           => DEV_MIR_ENABLE,
  C_OPB_AWIDTH               => C_OPB_AWIDTH,
  C_OPB_DWIDTH               => C_OPB_DWIDTH,
  C_FAMILY                   => C_FAMILY,
  C_IP_INTR_MODE_ARRAY       => IP_INTR_MODE_ARRAY,
  C_DEV_BURST_ENABLE         => DEV_BURST_ENABLE,
  C_INCLUDE_ADDR_CNTR        => INCLUDE_ADDR_CNTR,
  C_INCLUDE_WR_BUF           => INCLUDE_WR_BUF
)
port map
(
  OPB_select                  => OPB_select,
  OPB_DBus                    => OPB_DBus,
  OPB_ABus                    => OPB_ABus,
  OPB_BE                      => OPB_BE,
  OPB_RNW                     => OPB_RNW,
  OPB_seqAddr                 => OPB_seqAddr,
  Sln_DBus                    => Sln_DBus,
  Sln_xferAck                 => Sln_xferAck,
  Sln_errAck                  => Sln_errAck,
  Sln_retry                   => Sln_retry,
  Sln_toutSup                 => Sln_toutSup,
  Bus2IP_CS                   => open,
  Bus2IP_CE                   => open,
  Bus2IP_RdCE                 => iBus2IP_RdCE,
  Bus2IP_WrCE                 => iBus2IP_WrCE,
  Bus2IP_Data                 => iBus2IP_Data,
  Bus2IP_Addr                 => open,
  Bus2IP_AddrValid           => open,
  Bus2IP_BE                   => iBus2IP_BE,
  Bus2IP_RNW                  => open,
  Bus2IP_Burst                => open,
  IP2Bus_Data                 => iIP2Bus_Data,
  IP2Bus_Ack                  => iIP2Bus_Ack,
  IP2Bus_AddrAck              => '0',
  IP2Bus_Error                => iIP2Bus_Error,
  IP2Bus_Retry                => iIP2Bus_Retry,
  IP2Bus_ToutSup              => iIP2Bus_ToutSup,
  IP2Bus_PostedWrInh         => ENABLE_POSTED_WRITE,
  IP2RFIFO_Data               => ZERO_IP2RFIFO_Data,
  IP2RFIFO_WrMark             => '0',
  IP2RFIFO_WrRelease          => '0',
  IP2RFIFO_WrReq              => '0',
  IP2RFIFO_WrRestore          => '0',
  RFIFO2IP_AlmostFull        => open,
  RFIFO2IP_Full               => open,
  RFIFO2IP_Vacancy            => open,
  RFIFO2IP_WrAck              => open,
  IP2WFIFO_RdMark             => '0',

```

```

IP2WFIFO_RdRelease      => '0',
IP2WFIFO_RdReq         => '0',
IP2WFIFO_RdRestore     => '0',
WFIFO2IP_AlmostEmpty  => open,
WFIFO2IP_Data          => ZERO_WFIFO2IP_Data,
WFIFO2IP_Empty         => open,
WFIFO2IP_Occupancy    => open,
WFIFO2IP_RdAck        => open,
IP2Bus_IntrEvent      => iIP2Bus_IntrEvent,
IP2INTC_Irpt          => IP2INTC_Irpt,
Freeze                 => '0',
Bus2IP_Freeze         => open,
OPB_Clk                => OPB_Clk,
Bus2IP_Clk             => iBus2IP_Clk,
IP2Bus_Clk             => '0',
Reset                  => OPB_Rst,
Bus2IP_Reset          => iBus2IP_Reset
);

-----
-- instantiate the User Logic
-----
USER_LOGIC_I : entity opb_tmr_v1_02_a.user_logic
generic map
(
  -- MAP USER GENERICCS BELOW THIS LINE -----
  NUM_TMR                => NUM_TMR,
  -- MAP USER GENERICCS ABOVE THIS LINE -----

  C_DWIDTH              => USER_DWIDTH,
  C_NUM_CE               => USER_NUM_CE,
  C_IP_INTR_NUM         => USER_IP_INTR_NUM
)
port map
(
  -- MAP USER PORTS BELOW THIS LINE -----
  tmr2ppc_logic         => tmr2ppc_logic,
  tmr2ppc_min_voters   => tmr2ppc_min_voters,
  ppc2tmr_rst          => ppc2tmr_rst,
  ppc2tmr_bus_macro_en => ppc2tmr_bus_macro_en,
  -- MAP USER PORTS ABOVE THIS LINE -----

  Bus2IP_Clk           => iBus2IP_Clk,
  Bus2IP_Reset         => iBus2IP_Reset,
  IP2Bus_IntrEvent     => iIP2Bus_IntrEvent,
  Bus2IP_Data          => uBus2IP_Data,
  Bus2IP_BE            => uBus2IP_BE,
  Bus2IP_RdCE          => uBus2IP_RdCE,
  Bus2IP_WrCE          => uBus2IP_WrCE,
  IP2Bus_Data          => uIP2Bus_Data,
  IP2Bus_Ack           => iIP2Bus_Ack,
  IP2Bus_Retry         => iIP2Bus_Retry,
  IP2Bus_Error         => iIP2Bus_Error,
  IP2Bus_ToutSup       => iIP2Bus_ToutSup
);

-----
-- hooking up signal slicing
-----
uBus2IP_BE <= iBus2IP_BE(0 to USER_DWIDTH/8-1);
uBus2IP_Data <= iBus2IP_Data(0 to USER_DWIDTH-1);
uBus2IP_RdCE <= iBus2IP_RdCE(USER00_CE_INDEX to USER00_CE_INDEX+USER_NUM_CE-1);
uBus2IP_WrCE <= iBus2IP_WrCE(USER00_CE_INDEX to USER00_CE_INDEX+USER_NUM_CE-1);
iIP2Bus_Data(0 to USER_DWIDTH-1) <= uIP2Bus_Data;

end IMP;

```

## B.3: PowerPC Wrapper – User Logic – VHDL

```
-----
-- user_logic.vhd - entity/architecture pair
-----
--
-- *****
-- ** Copyright (c) 1995-2006 Xilinx, Inc. All rights reserved. **
-- ** ** **
-- ** Xilinx, Inc. **
-- ** XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" **
-- ** AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND **
-- ** SOLUTIONS FOR XILINX DEVICES. BY PROVIDING THIS DESIGN, CODE, **
-- ** OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, **
-- ** APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION **
-- ** THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT, **
-- ** AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE **
-- ** FOR YOUR IMPLEMENTATION. XILINX EXPRESSLY DISCLAIMS ANY **
-- ** WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE **
-- ** IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR **
-- ** REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF **
-- ** INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS **
-- ** FOR A PARTICULAR PURPOSE. **
-- ** ** **
-- *****
--
-----
-- Filename: user_logic.vhd
-- Version: 1.02.a
-- Description: User logic.
-- Date: Tue Apr 03 10:53:36 2007 (by Create and Import Peripheral Wizard)
-- VHDL Standard: VHDL'93
-----
-- Naming Conventions:
-- active low signals: *_n"
-- clock signals: "clk", "clk_div#", "clk_#x"
-- reset signals: "rst", "rst_n"
-- generics: "C_*"
-- user defined types: "**_TYPE"
-- state machine next state: "**_ns"
-- state machine current state: "**_cs"
-- combinatorial signals: "**_com"
-- pipelined or register delay signals: "**_d#"
-- counter signals: "**cnt*"
-- clock enable signals: "**_ce"
-- internal version of output port: "**_i"
-- device pins: "**_pin"
-- ports: "- Names begin with Uppercase"
-- processes: "**_PROCESS"
-- component instantiations: "<ENTITY_>I_<#|FUNC>"
-----
-- DO NOT EDIT BELOW THIS LINE -----
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library proc_common_v2_00_a;
use proc_common_v2_00_a.proc_common_pkg.all;
-- DO NOT EDIT ABOVE THIS LINE -----

--USER libraries added here

-----
-- Entity section
-----
-- Definition of Generics:
-- C_DWIDTH -- User logic data bus width
```

```

-- C_NUM_CE          -- User logic chip enable bus width
-- C_IP_INTR_NUM    -- User logic number of interrupt event
--
-- Definition of Ports:
-- Bus2IP_Clk       -- Bus to IP clock
-- Bus2IP_Reset     -- Bus to IP reset
-- IP2Bus_IntrEvent -- IP to Bus interrupt event
-- Bus2IP_Data      -- Bus to IP data bus for user logic
-- Bus2IP_BE        -- Bus to IP byte enables for user logic
-- Bus2IP_RdCE      -- Bus to IP read chip enable for user logic
-- Bus2IP_WrCE      -- Bus to IP write chip enable for user logic
-- IP2Bus_Data      -- IP to Bus data bus for user logic
-- IP2Bus_Ack       -- IP to Bus acknowledgement
-- IP2Bus_Retry     -- IP to Bus retry response
-- IP2Bus_Error     -- IP to Bus error response
-- IP2Bus_ToutSup   -- IP to Bus timeout suppress
-----

```

```

entity user_logic is
  generic

```

```

  (
    -- ADD USER GENERICS BELOW THIS LINE -----
    NUM_TMR : integer := 1;
    -- ADD USER GENERICS ABOVE THIS LINE -----

    -- DO NOT EDIT BELOW THIS LINE -----
    -- Bus protocol parameters, do not add to or delete
    C_DWIDTH : integer := 32;
    C_NUM_CE : integer := 2;
    C_IP_INTR_NUM : integer := 1
    -- DO NOT EDIT ABOVE THIS LINE -----
  );

```

```

port

```

```

  (
    -- ADD USER PORTS BELOW THIS LINE -----
    tmr2ppc_logic      : in  std_logic_vector(0 to 3*(NUM_TMR)-1);
    tmr2ppc_min_voters : in  std_logic_vector(0 to 3*(NUM_TMR)-1);
    ppc2tmr_rst        : out std_logic_vector(0 to 2);
    ppc2tmr_bus_macro_en : out std_logic_vector(0 to 2);
    -- ADD USER PORTS ABOVE THIS LINE -----

    -- DO NOT EDIT BELOW THIS LINE -----
    -- Bus protocol ports, do not add to or delete
    Bus2IP_Clk      : in  std_logic;
    Bus2IP_Reset    : in  std_logic;
    IP2Bus_IntrEvent : out std_logic_vector(0 to C_IP_INTR_NUM-1);
    Bus2IP_Data     : in  std_logic_vector(0 to C_DWIDTH-1);
    Bus2IP_BE       : in  std_logic_vector(0 to C_DWIDTH/8-1);
    Bus2IP_RdCE     : in  std_logic_vector(0 to C_NUM_CE-1);
    Bus2IP_WrCE     : in  std_logic_vector(0 to C_NUM_CE-1);
    IP2Bus_Data     : out std_logic_vector(0 to C_DWIDTH-1);
    IP2Bus_Ack      : out std_logic;
    IP2Bus_Retry    : out std_logic;
    IP2Bus_Error    : out std_logic;
    IP2Bus_ToutSup  : out std_logic
    -- DO NOT EDIT ABOVE THIS LINE -----
  );

```

```

end entity user_logic;

```

```

-----
-- Architecture section
-----

```

```

architecture IMP of user_logic is

```

```

  --USER signal declarations added here, as needed for user logic
  component Clk_Convrt
  port (
    Clk_in  : in  std_logic;
    Reset   : in  std_logic;
    Clk_out : out std_logic);

```

```

end component;

signal slow_clk      : std_logic;
signal slv_reg0_temp : std_logic_vector(0 to C_DWIDTH-1);

-----
-- Signals for user logic slave model s/w accessible register example
-----
signal slv_reg0      : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg1      : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg_write_select : std_logic_vector(0 to 1);
signal slv_reg_read_select  : std_logic_vector(0 to 1);
signal slv_ip2bus_data      : std_logic_vector(0 to C_DWIDTH-1);
signal slv_read_ack        : std_logic;
signal slv_write_ack       : std_logic;

-----
-- Signals for user logic interrupt example
-----
signal interrupt : std_logic_vector(0 to C_IP_INTR_NUM-1);

begin

--USER logic implementation added here

-----
-- Example code to read/write user logic slave model s/w accessible registers
--
-- Note:
-- The example code presented here is to show you one way of reading/writing
-- software accessible registers implemented in the user logic slave model.
-- Each bit of the Bus2IP_WrCE/Bus2IP_RdCE signals is configured to correspond
-- to one software accessible register by the top level template. For example,
-- if you have four 32 bit software accessible registers in the user logic, you
-- are basically operating on the following memory mapped registers:
--
--      Bus2IP_WrCE or   Memory Mapped
--      Bus2IP_RdCE     Register
--      "1000"          C_BASEADDR + 0x0
--      "0100"          C_BASEADDR + 0x4
--      "0010"          C_BASEADDR + 0x8
--      "0001"          C_BASEADDR + 0xc
--
-----
slv_reg_write_select <= Bus2IP_WrCE(0 to 1);
slv_reg_read_select  <= Bus2IP_RdCE(0 to 1);
slv_write_ack        <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1);
slv_read_ack         <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1);

-- Create an 80 Hz Clock signal
Clk_Convrt_1 : Clk_Convrt
  port map (
    Clk_in  => Bus2IP_Clk,
    Reset   => Bus2IP_Reset,
    Clk_out => slow_clk);

-- set the reset and enable bits as defined by the PPC Slave Register 1
ppc2tmr_rst      <= slv_reg1(29 to 31);
ppc2tmr_bus_macro_en <= slv_reg1(25 to 27);

-- this process loops through all of the bits of the logic and minority
-- voter outputs, and checks if there are any errors in the modules.
-- If so, it will set Slave Register 0 to the module in error that needs
-- to be reprogrammed.
error_reg : process (slow_clk)
  variable logic_temp      : std_logic_vector(0 to 2);
  variable min_voters_temp : std_logic_vector(0 to 2);
begin
  if slow_clk'event and slow_clk = '1' then
    if Bus2IP_Reset = '1' then
      slv_reg0_temp <= (others => '0');

```

```

    logic_temp      := (others => '0');
    min_voters_temp := (others => '0');
else
    if (slv_reg0 = x"00000000") then
        for i in 0 to NUM_TMR-1 loop
            -- find the logic and minority voter bit values corresponding
            -- to the current iteration
            logic_temp := tmr2ppc_logic(i) & tmr2ppc_logic(i + NUM_TMR)
                & tmr2ppc_logic(i + 2*NUM_TMR);
            min_voters_temp := tmr2ppc_min_voters(i)
                & tmr2ppc_min_voters(i + NUM_TMR)
                & tmr2ppc_min_voters(i + 2*NUM_TMR);

            case logic_temp(0 to 2) is
                when "001" =>
                    slv_reg0_temp <= x"00000003";
                when "010" =>
                    slv_reg0_temp <= x"00000002";
                when "100" =>
                    slv_reg0_temp <= x"00000001";
                when "110" =>
                    slv_reg0_temp <= x"00000003";
                when "101" =>
                    slv_reg0_temp <= x"00000002";
                when "011" =>
                    slv_reg0_temp <= x"00000001";
                when others =>
                    case min_voters_temp(0 to 2) is
                        when "001" =>
                            slv_reg0_temp <= x"00000003";
                        when "010" =>
                            slv_reg0_temp <= x"00000002";
                        when "100" =>
                            slv_reg0_temp <= x"00000001";
                        when others =>
                            end case;
                    end case;

            end loop;
        else
            slv_reg0_temp <= (others => '0');
        end if;
    end if;
end process error_reg;

-- implement slave model register(s)
SLAVE_REG_WRITE_PROC : process(Bus2IP_Clk) is
begin
    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
        if Bus2IP_Reset = '1' then
            slv_reg0 <= (others => '0');
            slv_reg1 <= (others => '0');
        else
            if (slv_reg_write_select = "10") then
                for byte_index in 0 to (C_DWIDTH/8)-1 loop
                    if (Bus2IP_BE(byte_index) = '1') then
                        slv_reg0(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to
                            byte_index*8+7);
                    end if;
                end loop;
            elsif (slv_reg_write_select = "01") then
                for byte_index in 0 to (C_DWIDTH/8)-1 loop
                    if (Bus2IP_BE(byte_index) = '1') then
                        slv_reg1(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to
                            byte_index*8+7);
                    end if;
                end loop;
            elsif (slv_reg0 = x"00000000") then
                slv_reg0 <= slv_reg0_temp;
            end if;
        end if;
    end if;
end process;

```

```

        end if;
    end if;
end process SLAVE_REG_WRITE_PROC;

-- implement slave model register read mux
SLAVE_REG_READ_PROC : process(slv_reg_read_select, slv_reg0, slv_reg1) is
begin

    case slv_reg_read_select is
        when "10"    => slv_ip2bus_data <= slv_reg0;
        when "01"    => slv_ip2bus_data <= slv_reg1;
        when others => slv_ip2bus_data <= (others => '0');
    end case;

end process SLAVE_REG_READ_PROC;

-----
-- Example code to generate user logic interrupts
--
-- Note:
-- The example code presented here is to show you one way of generating
-- interrupts from the user logic. This code snippet infers a counter
-- and generate the interrupts whenever the counter rollover (the counter
-- will rollover ~21 sec @50Mhz).
-----
INTR_PROC : process(Bus2IP_Clk) is
    constant COUNT_SIZE : integer                := 30;
    constant ALL_ONES   : std_logic_vector(0 to COUNT_SIZE-1) := (others => '1');
    variable counter    : std_logic_vector(0 to COUNT_SIZE-1);
begin

    if (Bus2IP_Clk'event and Bus2IP_Clk = '1') then
        if (Bus2IP_Reset = '1') then
            counter := (others => '0');
            interrupt <= (others => '0');
        else
            counter := counter + 1;
            if (counter = ALL_ONES) then
                interrupt <= (others => '1');
            else
                interrupt <= (others => '0');
            end if;
        end if;
    end if;

end process INTR_PROC;

IP2Bus_IntrEvent <= interrupt;

-----
-- Example code to drive IP to Bus signals
-----
IP2Bus_Data <= slv_ip2bus_data;

IP2Bus_Ack    <= slv_write_ack or slv_read_ack;
IP2Bus_Error  <= '0';
IP2Bus_Retry  <= '0';
IP2Bus_ToutSup <= '0';

end IMP;

```

## B.4: PowerPC Wrapper – Clock Converter – VHDL

```
-----  
-----  
-- Worcester Polytechnic Institute  
-- General Dynamics C4 Systems MQP  
-- Evan Custodio  
-- Brian Marsland  
-- D Term 2007  
-- Self-Healing Partial Reconfiguration of an FPGA  
-- Filename: Clk_Convrt.vhd  
-----  
-----  
-- This module takes in a 100 Mhz clock and outputs a 80 Hz clock signal. The  
-- code is borrowed and modified from the ECE 3801 Online Laboratory Resources.  
-- The ECE 3801 site is viewable at http://ece.wpi.edu/courses/ee3801/  
-----  
-----  
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.STD_LOGIC_ARITH.all;  
use IEEE.STD_LOGIC_UNSIGNED.all;  
  
entity Clk_Convrt is  
    port (Clk_in   : in  std_logic;  
          Reset    : in  std_logic;  
          Clk_out  : out std_logic  
        );  
end Clk_Convrt;  
  
architecture Behavioral of Clk_Convrt is  
    signal tmp_clk : std_logic;  
begin  
    Clk_out <= tmp_clk;  
  
    -- create an 80 Hz clock  
    process(Reset, Clk_in)  
        variable counter : integer range 0 to 625_000;  
    begin  
        if Reset = '1' then  
            counter := 0;  
            tmp_clk <= '0';  
        elsif Clk_in'event and Clk_in = '1' then  
            counter := counter + 1;  
  
            if counter = 625_000 then  
                tmp_clk <= not tmp_clk;  
                counter := 0;  
            end if;  
        end if;  
    end process;  
end Behavioral;
```



## B.5: PowerPC System – VHDL

Due to the length of this file, only the modifications to the automatically generated system.vhd file will be shown here. The modifications are bolded in the following code. This file is located in <EDK\_project>\hdl\system.vhd.

```
-----  
-- system.vhd  
-----  
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
  
library UNISIM;  
use UNISIM.VCOMPONENTS.ALL;  
  
library dcm_module_v1_00_a;  
use dcm_module_v1_00_a.All;  
  
entity system is  
  port (  
    ...  
  );  
  
  attribute incremental_synthesis : STRING;  
  attribute incremental_synthesis of system: entity is "yes";  
  attribute x_core_info : STRING;  
  attribute x_core_info of system: entity is "dcm_module_v1_00_a";  
end system;  
  
architecture STRUCTURE of system is  
  
  ...  
  
  component sram_256kx32_flash_lmx32_util_bus_split_1_wrapper is  
    port (  
      Sig : in std_logic_vector(0 to 31);  
      Out1 : out std_logic_vector(9 to 29);  
      Out2 : out std_logic_vector(30 to 31)  
    );  
  end component;  
  
  attribute box_type of sram_256kx32_flash_lmx32_util_bus_split_1_wrapper: component is  
  "black_box";  
  
  component dcm_module is  
    generic (  
      C_DFS_FREQUENCY_MODE : STRING;  
      C_DLL_FREQUENCY_MODE : STRING;  
      C_DUTY_CYCLE_CORRECTION : BOOLEAN;  
      C_CLKIN_DIVIDE_BY_2 : BOOLEAN;  
      C_CLK_FEEDBACK : STRING;  
      C_CLKOUT_PHASE_SHIFT : STRING;  
      C_DSS_MODE : STRING;  
      C_STARTUP_WAIT : BOOLEAN;  
      C_PHASE_SHIFT : INTEGER;  
      C_CLKFX_MULTIPLY : INTEGER;  
      C_CLKFX_DIVIDE : INTEGER;  
      C_CLKDV_DIVIDE : REAL;  
      C_CLKIN_PERIOD : REAL;  
      C_DESKEW_ADJUST : STRING;  
      C_CLKIN_BUF : BOOLEAN;  
      C_CLKFB_BUF : BOOLEAN;  
      C_CLK0_BUF : BOOLEAN;  
    )  
  end component;
```

```

    C_CLK90_BUF : BOOLEAN;
    C_CLK180_BUF : BOOLEAN;
    C_CLK270_BUF : BOOLEAN;
    C_CLKDV_BUF : BOOLEAN;
    C_CLK2X_BUF : BOOLEAN;
    C_CLK2X180_BUF : BOOLEAN;
    C_CLKFX_BUF : BOOLEAN;
    C_CLKFX180_BUF : BOOLEAN;
    C_EXT_RESET_HIGH : integer;
    C_FAMILY : string
);
port (
    RST : in std_logic;
    CLKIN : in std_logic;
    CLKFB : in std_logic;
    PSEN : in std_logic;
    PSINCDEC : in std_logic;
    PSCLK : in std_logic;
    DSSEN : in std_logic;
    CLK0 : out std_logic;
    CLK90 : out std_logic;
    CLK180 : out std_logic;
    CLK270 : out std_logic;
    CLKDV : out std_logic;
    CLK2X : out std_logic;
    CLK2X180 : out std_logic;
    CLKFX : out std_logic;
    CLKFX180 : out std_logic;
    STATUS : out std_logic_vector(7 downto 0);
    LOCKED : out std_logic;
    PSDONE : out std_logic
);
end component;

component opb_timer_1_wrapper is
port (
    OPB_Clk : in std_logic;
    OPB_Rst : in std_logic;
    OPB_ABus : in std_logic_vector(0 to 31);
    OPB_BE : in std_logic_vector(0 to 3);
    OPB_DBus : in std_logic_vector(0 to 31);
    OPB_RNW : in std_logic;
    OPB_select : in std_logic;
    OPB_seqAddr : in std_logic;
    TC_DBus : out std_logic_vector(0 to 31);
    TC_errAck : out std_logic;
    TC_retry : out std_logic;
    TC_toutSup : out std_logic;
    TC_xferAck : out std_logic;
    CaptureTrig0 : in std_logic;
    CaptureTrig1 : in std_logic;
    GenerateOut0 : out std_logic;
    GenerateOut1 : out std_logic;
    PWM0 : out std_logic;
    Interrupt : out std_logic;
    Freeze : in std_logic
);
end component;

attribute box_type of opb_timer_1_wrapper: component is "black_box";

...

begin

...

sram_256kx32_flash_lmx32_util_bus_split_1 : sram_256kx32_flash_lmx32_util_bus_split_1_wrapper
port map (
    Sig => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_split,
    Out1 => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A,

```

```

    Out2 => open
);

dcm_0 : dcm_module
generic map (
    C_DFS_FREQUENCY_MODE => "LOW",
    C_DLL_FREQUENCY_MODE => "LOW",
    C_DUTY_CYCLE_CORRECTION => TRUE,
    C_CLKIN_DIVIDE_BY_2 => FALSE,
    C_CLK_FEEDBACK => "1X",
    C_CLKOUT_PHASE_SHIFT => "NONE",
    C_DSS_MODE => "NONE",
    C_STARTUP_WAIT => FALSE,
    C_PHASE_SHIFT => 0,
    C_CLKFX_MULTIPLY => 4,
    C_CLKFX_DIVIDE => 1,
    C_CLKDV_DIVIDE => 2.0,
    C_CLKIN_PERIOD => 10.000000,
    C_DESKEW_ADJUST => "SYSTEM_SYNCHRONOUS",
    C_CLKIN_BUF => FALSE,
    C_CLKFB_BUF => FALSE,
    C_CLK0_BUF => TRUE,
    C_CLK90_BUF => FALSE,
    C_CLK180_BUF => FALSE,
    C_CLK270_BUF => FALSE,
    C_CLKDV_BUF => FALSE,
    C_CLK2X_BUF => FALSE,
    C_CLK2X180_BUF => FALSE,
    C_CLKFX_BUF => FALSE,
    C_CLKFX180_BUF => FALSE,
    C_EXT_RESET_HIGH => 1,
    C_FAMILY => "virtex2p"
)
port map (
    RST => net_gnd0,
    CLKIN => dcm_clk_s,
    CLKFB => sys_clk_s,
    PSEN => net_gnd0,
    PSINCDEC => net_gnd0,
    PSClk => net_gnd0,
    DSSSEN => net_gnd0,
    CLK0 => sys_clk_s,
    CLK90 => open,
    CLK180 => open,
    CLK270 => open,
    CLKDV => open,
    CLK2X => open,
    CLK2X180 => open,
    CLKFX => open,
    CLKFX180 => open,
    STATUS => open,
    LOCKED => dcm_0_lock,
    PSDONE => open
);

opb_timer_1 : opb_timer_1_wrapper
port map (
    OPB_Clk => sys_clk_s,
    OPB_Rst => opb_OPB_Rst,
    OPB_ABus => opb_OPB_ABus,
    OPB_BE => opb_OPB_BE,
    OPB_DBus => opb_OPB_DBus,
    OPB_RNW => opb_OPB_RNW,
    OPB_select => opb_OPB_select,
    OPB_seqAddr => opb_OPB_seqAddr,
    TC_DBus => opb_Sl_DBus(96 to 127),
    TC_errAck => opb_Sl_errAck(3),
    TC_retry => opb_Sl_retry(3),
    TC_toutSup => opb_Sl_toutSup(3),
    TC_xferAck => opb_Sl_xferAck(3),
    CaptureTrig0 => net_gnd0,

```

```

        CaptureTrig1 => net_gnd0,
        GenerateOut0 => open,
        GenerateOut1 => open,
        PWM0 => open,
        Interrupt => opb_timer_1_Interrupt(0),
        Freeze => net_gnd0
    );

    ...

end architecture STRUCTURE;

```

## B.6: PowerPC System Synthesis Parameters

In the file <EDK\_project>\synthesis\system\_xst.prj:

```

VHDL dcm_module_v1_00_a
C:\EDK\hw\XilinxProcessorIPLib\pcores\dcm_module_v1_00_a/hdl/vhdl/dcm_module.vhd
vhdl work ../hdl/system.vhd

```

## B.7: Top-Level – VHDL

```

-----
-- Worcester Polytechnic Institute
-- General Dynamics C4 Systems MQP
-- Evan Custodio
-- Brian Marsland
-- D Term 2007
-- Self-Healing Partial Reconfiguration of an FPGA
-- Filename: top_level_tmr.vhd
-----

-- Top-level design for the self-healing system. This file is designed
-- generically based on NUM_TMR such that the routing and bus macros needed to
-- connect the specified number of voting modules are instantiated
-- automatically.
--
-- This file must be synthesized with I/O Buffer and Keep Hierarchy.
-- All modules instantiated must be black boxes.
-----

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use busmacro_xc2vp_pkg.all;

entity top_level_tmr is
    generic (
        NUM_TMR                : integer := 1);
    port (
        CLK                    : in     std_logic;
        RST                    : in     std_logic;
        error_in                : in     std_logic_vector(0 to 2);
        tmr2ppc_logic           : out    std_logic_vector(0 to ((3*NUM_TMR)-1));
        tmr2ppc_min_voters      : out    std_logic_vector(0 to ((3*NUM_TMR)-1));
        X                       : out    std_logic;

```

```

fpga_0_RS232_1_RX_pin      : in      std_logic;
fpga_0_RS232_1_TX_pin      : out      std_logic;
fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin      : inout  std_logic_vector(0 to 31);
fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin     : out      std_logic_vector(0 to 11);
fpga_0_SDRAM_8Mx32_1_SDRAM_DQM_pin     : out      std_logic_vector(0 to 3);
fpga_0_SDRAM_8Mx32_1_SDRAM_WEn_pin     : out      std_logic;
fpga_0_SDRAM_8Mx32_1_SDRAM_CKE_pin     : out      std_logic;
fpga_0_SDRAM_8Mx32_1_SDRAM_CS_n_pin    : out      std_logic;
fpga_0_SDRAM_8Mx32_1_SDRAM_CAS_n_pin   : out      std_logic;
fpga_0_SDRAM_8Mx32_1_SDRAM_RAS_n_pin   : out      std_logic;
fpga_0_SDRAM_8Mx32_1_SDRAM_Clk_pin     : out      std_logic;
fpga_0_SDRAM_8Mx32_1_SDRAM_BankAddr_pin : out      std_logic_vector(0 to 1);
fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin : out      std_logic_vector(9 to 29);
fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin : inout  std_logic_vector(0 to 31);
fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin : out      std_logic_vector(0 to 3);
fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_WEN_pin : out      std_logic;
fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_OEN_pin : out      std_logic_vector(0 to 1);
fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_CEN_pin : out      std_logic_vector(0 to 1);
fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_RPN_pin : out      std_logic;
end top_level_tmr;

```

architecture Behavioral of top\_level\_tmr is

```

component logic_and_voters
generic (
    NUM_TMR      : integer);
port (
    error_in      : in  std_logic;
    enable_1_n    : in  std_logic;
    enable_2_n    : in  std_logic;
    logic_in_1    : in  std_logic_vector(0 to NUM_TMR-1);
    logic_in_2    : in  std_logic_vector(0 to NUM_TMR-1);
    maj_voter_in_1 : in  std_logic_vector(0 to NUM_TMR-1);
    maj_voter_in_2 : in  std_logic_vector(0 to NUM_TMR-1);
    feedback_in   : in  std_logic;
    CLK           : in  std_logic;
    RST           : in  std_logic;
    majority_out  : out std_logic_vector(0 to NUM_TMR-1);
    minority_out  : out std_logic_vector(0 to NUM_TMR-1);
    logic_out     : out std_logic_vector(0 to NUM_TMR-1);
    voter_out     : out std_logic_vector(0 to NUM_TMR-1));
end component;

```

```

component safe_module
generic (
    NUM_TMR : integer);
port (
    voter_1 : in  std_logic_vector(0 to NUM_TMR-1);
    voter_2 : in  std_logic_vector(0 to NUM_TMR-1);
    voter_3 : in  std_logic_vector(0 to NUM_TMR-1);
    enable_n : in  std_logic_vector(0 to 2);
    X        : out std_logic_vector(0 to NUM_TMR-1));
end component;

```

```

component system
port (
    fpga_0_RS232_1_RX_pin      : in      std_logic;
    fpga_0_RS232_1_TX_pin      : out      std_logic;
    fpga_0_LEDs_8Bit_GPIO_d_out_pin : out      std_logic_vector(0 to 7);
    fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin      : inout  std_logic_vector(0 to 31);
    fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin     : out      std_logic_vector(0 to 11);
    fpga_0_SDRAM_8Mx32_1_SDRAM_DQM_pin     : out      std_logic_vector(0 to 3);
    fpga_0_SDRAM_8Mx32_1_SDRAM_WEn_pin     : out      std_logic;
    fpga_0_SDRAM_8Mx32_1_SDRAM_CKE_pin     : out      std_logic;
    fpga_0_SDRAM_8Mx32_1_SDRAM_CS_n_pin    : out      std_logic;
    fpga_0_SDRAM_8Mx32_1_SDRAM_CAS_n_pin   : out      std_logic;
    fpga_0_SDRAM_8Mx32_1_SDRAM_RAS_n_pin   : out      std_logic;
    fpga_0_SDRAM_8Mx32_1_SDRAM_Clk_pin     : out      std_logic;
    fpga_0_SDRAM_8Mx32_1_SDRAM_BankAddr_pin : out      std_logic_vector(0 to 1);
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin : out      std_logic_vector(9 to 29);
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin : inout  std_logic_vector(0 to 31);

```

```

    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin : out    std_logic_vector(0 to 3);
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_WEN_pin : out    std_logic;
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_OEN_pin : out    std_logic_vector(0 to 1);
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_CEN_pin : out    std_logic_vector(0 to 1);
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_RPN_pin : out    std_logic;
    sys_clk_pin                                 : in      std_logic;
    sys_rst_pin                                 : in      std_logic;
    opb_tmr_0_tmr2ppc_logic_pin                : in      std_logic_vector(0 to ((3*NUM_TMR)-1));
--2);
    opb_tmr_0_tmr2ppc_min_voters_pin          : in      std_logic_vector(0 to ((3*NUM_TMR)-1));
--2);
    opb_tmr_0_ppc2tmr_rst_pin                  : out    std_logic_vector(0 to 2);
    opb_tmr_0_ppc2tmr_bus_macro_en_pin        : out    std_logic_vector(0 to 2)
);
end component;

signal feedback_in_signal                    : std_logic_vector(0 to 2);
signal rst_signal                            : std_logic_vector(0 to 2);
signal error_in_signal                       : std_logic_vector(0 to 2);
signal logic_out_0_signal                    : std_logic_vector(0 to NUM_TMR-1);
signal logic_out_1_signal                    : std_logic_vector(0 to NUM_TMR-1);
signal logic_out_2_signal                    : std_logic_vector(0 to NUM_TMR-1);
signal logic_0_signal                        : std_logic_vector(0 to NUM_TMR-1);
signal logic_1_signal                        : std_logic_vector(0 to NUM_TMR-1);
signal logic_2_signal                        : std_logic_vector(0 to NUM_TMR-1);
signal logic_in_1_signal_0                   : std_logic_vector(0 to NUM_TMR-1);
signal logic_in_2_signal_0                   : std_logic_vector(0 to NUM_TMR-1);
signal logic_in_0_signal_1                   : std_logic_vector(0 to NUM_TMR-1);
signal logic_in_2_signal_1                   : std_logic_vector(0 to NUM_TMR-1);
signal logic_in_0_signal_2                   : std_logic_vector(0 to NUM_TMR-1);
signal logic_in_1_signal_2                   : std_logic_vector(0 to NUM_TMR-1);
signal majority_out_0_signal                 : std_logic_vector(0 to NUM_TMR-1);
signal majority_out_1_signal                 : std_logic_vector(0 to NUM_TMR-1);
signal majority_out_2_signal                 : std_logic_vector(0 to NUM_TMR-1);
signal majority_0_signal                     : std_logic_vector(0 to NUM_TMR-1);
signal majority_1_signal                     : std_logic_vector(0 to NUM_TMR-1);
signal majority_2_signal                     : std_logic_vector(0 to NUM_TMR-1);
signal majority_in_1_signal_0                : std_logic_vector(0 to NUM_TMR-1);
signal majority_in_2_signal_0                : std_logic_vector(0 to NUM_TMR-1);
signal majority_in_0_signal_1                : std_logic_vector(0 to NUM_TMR-1);
signal majority_in_2_signal_1                : std_logic_vector(0 to NUM_TMR-1);
signal majority_in_0_signal_2                : std_logic_vector(0 to NUM_TMR-1);
signal majority_in_1_signal_2                : std_logic_vector(0 to NUM_TMR-1);
signal minority_out_0_signal                 : std_logic_vector(0 to NUM_TMR-1);
signal minority_out_1_signal                 : std_logic_vector(0 to NUM_TMR-1);
signal minority_out_2_signal                 : std_logic_vector(0 to NUM_TMR-1);
signal minority_0_signal                     : std_logic_vector(0 to NUM_TMR-1);
signal minority_1_signal                     : std_logic_vector(0 to NUM_TMR-1);
signal minority_2_signal                     : std_logic_vector(0 to NUM_TMR-1);
signal voter_out_0_signal                    : std_logic_vector(0 to NUM_TMR-1);
signal voter_out_1_signal                    : std_logic_vector(0 to NUM_TMR-1);
signal voter_out_2_signal                    : std_logic_vector(0 to NUM_TMR-1);
signal voter_0_signal                        : std_logic_vector(0 to NUM_TMR-1);
signal voter_1_signal                        : std_logic_vector(0 to NUM_TMR-1);
signal voter_2_signal                        : std_logic_vector(0 to NUM_TMR-1);
signal X_signal                              : std_logic_vector(0 to NUM_TMR-1);
signal tmr2ppc_logic_signal                  : std_logic_vector(0 to ((3*NUM_TMR)-1));
signal tmr2ppc_min_voters_signal             : std_logic_vector(0 to ((3*NUM_TMR)-1));
signal ppc2tmr_rst_signal                    : std_logic_vector(0 to 2);
signal ppc2tmr_bus_macro_en_signal           : std_logic_vector(0 to 2);
signal enable_1_n_signal_0                   : std_logic;
signal enable_2_n_signal_0                   : std_logic;
signal enable_0_n_signal_1                   : std_logic;
signal enable_2_n_signal_1                   : std_logic;
signal enable_0_n_signal_2                   : std_logic;
signal enable_1_n_signal_2                   : std_logic;

begin

    -- instantiate the first partial module containing logic module and voters
    logic_and_voters_0 : logic_and_voters

```

```

generic map (
  NUM_TMR      => NUM_TMR)
port map (
  error_in      => error_in_signal(0),
  enable_1_n    => enable_1_n_signal_0,
  enable_2_n    => enable_2_n_signal_0,
  logic_in_1    => logic_in_1_signal_0,
  logic_in_2    => logic_in_2_signal_0,
  maj_voter_in_1 => majority_in_1_signal_0,
  maj_voter_in_2 => majority_in_2_signal_0,
  feedback_in   => feedback_in_signal(0),
  CLK           => CLK,
  RST           => rst_signal(0),
  majority_out  => majority_out_0_signal,
  minority_out  => minority_out_0_signal,
  logic_out     => logic_out_0_signal,
  voter_out    => voter_out_0_signal);

-- instantiate the second partial module containing logic module and voters
logic_and_voters_1 : logic_and_voters
generic map (
  NUM_TMR      => NUM_TMR)
port map (
  error_in      => error_in_signal(1),
  enable_1_n    => enable_0_n_signal_1,
  enable_2_n    => enable_2_n_signal_1,
  logic_in_1    => logic_in_0_signal_1,
  logic_in_2    => logic_in_2_signal_1,
  maj_voter_in_1 => majority_in_0_signal_1,
  maj_voter_in_2 => majority_in_2_signal_1,
  feedback_in   => feedback_in_signal(1),
  CLK           => CLK,
  RST           => rst_signal(1),
  majority_out  => majority_out_1_signal,
  minority_out  => minority_out_1_signal,
  logic_out     => logic_out_1_signal,
  voter_out    => voter_out_1_signal);

-- instantiate the third partial module containing logic modules and voters
logic_and_voters_2 : logic_and_voters
generic map (
  NUM_TMR      => NUM_TMR)
port map (
  error_in      => error_in_signal(2),
  enable_1_n    => enable_0_n_signal_2,
  enable_2_n    => enable_1_n_signal_2,
  logic_in_1    => logic_in_0_signal_2,
  logic_in_2    => logic_in_1_signal_2,
  maj_voter_in_1 => majority_in_0_signal_2,
  maj_voter_in_2 => majority_in_1_signal_2,
  feedback_in   => feedback_in_signal(2),
  CLK           => CLK,          --clk_signal(2),
  RST           => rst_signal(2),
  majority_out  => majority_out_2_signal,
  minority_out  => minority_out_2_signal,
  logic_out     => logic_out_2_signal,
  voter_out    => voter_out_2_signal);

-- instantiate the safe module, which ORs the final voter signals from the
-- three partial modules
safe_module_1 : safe_module
generic map (
  NUM_TMR      => NUM_TMR)
port map (
  voter_1      => voter_0_signal,
  voter_2      => voter_1_signal,
  voter_3      => voter_2_signal,
  enable_n     => ppc2tmr_bus_macro_en_signal,
  X            => X_signal);

-- the first (and only) bit of the output of the safe module

```

```

X <= X_signal(0);

-- generate Input Bus Macros for each of the three partial modules that
-- includes the feedback, reset, and input error signals
input_bus_macros_all : for i in 0 to 2 generate
  inputbusleft      : busmacro_xc2vp_l2r_async_narrow
  port map (
    input0 => X_signal(0), output0 => feedback_in_signal(i),
    input1 => '0',                --CLK, output1 => clk_signal(i),
    input2 => ppc2tmr_rst_signal(i), output2 => rst_signal(i),
    input3 => error_in(i), output3 => error_in_signal(i),
    input4 => '0',
    input5 => '0',
    input6 => '0',
    input7 => '0');
end generate input_bus_macros_all;

-- generate Input Bus Macro for the enable signals input to the first partial
-- module
inputbusleft2_0 : busmacro_xc2vp_l2r_async_narrow
  port map (
    input0 => ppc2tmr_bus_macro_en_signal(1), output0 => enable_1_n_signal_0,
    input1 => ppc2tmr_bus_macro_en_signal(2), output1 => enable_2_n_signal_0,
    input2 => '0',
    input3 => '0',
    input4 => '0',
    input5 => '0',
    input6 => '0',
    input7 => '0');

-- generate Input Bus Macro for the enable signals input to the second partial
-- module
inputbusleft2_1 : busmacro_xc2vp_l2r_async_narrow
  port map (
    input0 => ppc2tmr_bus_macro_en_signal(0), output0 => enable_0_n_signal_1,
    input1 => ppc2tmr_bus_macro_en_signal(2), output1 => enable_2_n_signal_1,
    input2 => '0',
    input3 => '0',
    input4 => '0',
    input5 => '0',
    input6 => '0',
    input7 => '0');

-- generate Input Bus Macro for the enable signals input to the third partial
-- module
inputbusleft2_2 : busmacro_xc2vp_l2r_async_narrow
  port map (
    input0 => ppc2tmr_bus_macro_en_signal(0), output0 => enable_0_n_signal_2,
    input1 => ppc2tmr_bus_macro_en_signal(1), output1 => enable_1_n_signal_2,
    input2 => '0',
    input3 => '0',
    input4 => '0',
    input5 => '0',
    input6 => '0',
    input7 => '0');

-- generate all of the Input Bus Macros necessary for the voting circuitry of
-- the first partial module
input_bus_macros_0 : for i in 0 to NUM_TMR-1 generate
  inputbusleft_0   : busmacro_xc2vp_l2r_async_narrow
  port map (
    input0 => logic_1_signal(i), output0 => logic_in_1_signal_0(i),
    input1 => logic_2_signal(i), output1 => logic_in_2_signal_0(i),
    input2 => majority_1_signal(i), output2 => majority_in_1_signal_0(i),
    input3 => majority_2_signal(i), output3 => majority_in_2_signal_0(i),
    input4 => '0',
    input5 => '0',
    input6 => '0',
    input7 => '0');
end generate input_bus_macros_0;

```



```

-- generate all of the Output Bus Macros necessary for logic and voter
-- outputs of the first partial module
output_bus_macros_0 : for i in 0 to NUM_TMR-1 generate
  outputbusright_0 : busmacro_xc2vp_l2r_async_narrow
    port map (
      input0 => logic_out_0_signal(i), output0 => logic_0_signal(i),
      input1 => majority_out_0_signal(i), output1 => majority_0_signal(i),
      input2 => minority_out_0_signal(i), output2 => minority_0_signal(i),
      input3 => voter_out_0_signal(i), output3 => voter_0_signal(i),
      input4 => '0',
      input5 => '0',
      input6 => '0',
      input7 => '0');
end generate output_bus_macros_0;

-- generate all of the Input Bus Macros necessary for the voting circuitry of
-- the second partial module
input_bus_macros_1 : for i in 0 to NUM_TMR-1 generate
  inputbusleft_1 : busmacro_xc2vp_l2r_async_narrow
    port map (
      input0 => logic_0_signal(i), output0 => logic_in_0_signal_1(i),
      input1 => logic_2_signal(i), output1 => logic_in_2_signal_1(i),
      input2 => majority_0_signal(i), output2 => majority_in_0_signal_1(i),
      input3 => majority_2_signal(i), output3 => majority_in_2_signal_1(i),
      input4 => '0',
      input5 => '0',
      input6 => '0',
      input7 => '0');
end generate input_bus_macros_1;

-- generate all of the Output Bus Macros necessary for logic and voter
-- outputs of the second partial module
output_bus_macros_1 : for i in 0 to NUM_TMR-1 generate
  outputbusright_1 : busmacro_xc2vp_l2r_async_narrow
    port map (
      input0 => logic_out_1_signal(i), output0 => logic_1_signal(i),
      input1 => majority_out_1_signal(i), output1 => majority_1_signal(i),
      input2 => minority_out_1_signal(i), output2 => minority_1_signal(i),
      input3 => voter_out_1_signal(i), output3 => voter_1_signal(i),
      input4 => '0',
      input5 => '0',
      input6 => '0',
      input7 => '0');
end generate output_bus_macros_1;

-- generate all of the Input Bus Macros necessary for the voting circuitry of
-- the third partial module
input_bus_macros_2 : for i in 0 to NUM_TMR-1 generate
  inputbusleft_2 : busmacro_xc2vp_l2r_async_narrow
    port map (
      input0 => logic_0_signal(i), output0 => logic_in_0_signal_2(i),
      input1 => logic_1_signal(i), output1 => logic_in_1_signal_2(i),
      input2 => majority_0_signal(i), output2 => majority_in_0_signal_2(i),
      input3 => majority_1_signal(i), output3 => majority_in_1_signal_2(i),
      input4 => '0',
      input5 => '0',
      input6 => '0',
      input7 => '0');
end generate input_bus_macros_2;

-- generate all of the Output Bus Macros necessary for logic and voter
-- outputs of the third partial module
output_bus_macros_2 : for i in 0 to NUM_TMR-1 generate
  outputbusright_2 : busmacro_xc2vp_l2r_async_narrow
    port map (
      input0 => logic_out_2_signal(i), output0 => logic_2_signal(i),
      input1 => majority_out_2_signal(i), output1 => majority_2_signal(i),
      input2 => minority_out_2_signal(i), output2 => minority_2_signal(i),
      input3 => voter_out_2_signal(i), output3 => voter_2_signal(i),
      input4 => '0',
      input5 => '0',

```

```

        input6 => '0',
        input7 => '0');
end generate output_bus_macros_2;

-- concatenate the logic and minority voter signals for the PowerPC
tmr2ppc_logic_signal      <= logic_0_signal & logic_1_signal & logic_2_signal;
tmr2ppc_min_voters_signal <= minority_0_signal & minority_1_signal & minority_2_signal;

-- also output these signals to the LEDs
tmr2ppc_logic           <= tmr2ppc_logic_signal;
tmr2ppc_min_voters     <= tmr2ppc_min_voters_signal;

-- instantiate the black box module for the PowerPC system
ppc_system : system
  port map (
    fpga_0_RS232_1_RX_pin      => fpga_0_RS232_1_RX_pin,
    fpga_0_RS232_1_TX_pin      => fpga_0_RS232_1_TX_pin,
    fpga_0_LEDs_8Bit_GPIO_d_out_pin => open,
    fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin      => fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin,
    fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin     => fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin,
    fpga_0_SDRAM_8Mx32_1_SDRAM_DQM_pin     => fpga_0_SDRAM_8Mx32_1_SDRAM_DQM_pin,
    fpga_0_SDRAM_8Mx32_1_SDRAM_WEn_pin     => fpga_0_SDRAM_8Mx32_1_SDRAM_WEn_pin,
    fpga_0_SDRAM_8Mx32_1_SDRAM_CKE_pin     => fpga_0_SDRAM_8Mx32_1_SDRAM_CKE_pin,
    fpga_0_SDRAM_8Mx32_1_SDRAM_CS_n_pin    => fpga_0_SDRAM_8Mx32_1_SDRAM_CS_n_pin,
    fpga_0_SDRAM_8Mx32_1_SDRAM_CAS_n_pin   => fpga_0_SDRAM_8Mx32_1_SDRAM_CAS_n_pin,
    fpga_0_SDRAM_8Mx32_1_SDRAM_RAS_n_pin   => fpga_0_SDRAM_8Mx32_1_SDRAM_RAS_n_pin,
    fpga_0_SDRAM_8Mx32_1_SDRAM_Clk_pin     => fpga_0_SDRAM_8Mx32_1_SDRAM_Clk_pin,
    fpga_0_SDRAM_8Mx32_1_SDRAM_BankAddr_pin => fpga_0_SDRAM_8Mx32_1_SDRAM_BankAddr_pin,
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin,
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin,
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin,
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_WEN_pin => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_WEN_pin,
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_OEN_pin => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_OEN_pin,
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_CEN_pin => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_CEN_pin,
    fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_RPN_pin => fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_RPN_pin,
    sys_clk_pin                => CLK,
    sys_rst_pin                => RST,
    opb_tmr_0_tmr2ppc_logic_pin => tmr2ppc_logic_signal,
    opb_tmr_0_tmr2ppc_min_voters_pin => tmr2ppc_min_voters_signal,
    opb_tmr_0_ppc2tmr_rst_pin   => ppc2tmr_rst_signal,
    opb_tmr_0_ppc2tmr_bus_macro_en_pin => ppc2tmr_bus_macro_en_signal
  );
end Behavioral;

```

## B.8: Reconfigurable Module – Logic and Voters – VHDL

```

-----
-- Worcester Polytechnic Institute
-- General Dynamics C4 Systems MQP
-- Evan Custodio
-- Brian Marsland
-- D Term 2007
-- Self-Healing Partial Reconfiguration of an FPGA
-- Filename: logic_and_voters.vhd
-----

-- This module is the top-level partial module. It includes the logic module
-- that is being replicated as part of the TMR design. It also instantiates
-- correc number of majority and minority voters based on the NUM_TMR generic.
-----

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

```

```

use IEEE.STD_LOGIC_UNSIGNED.all;

entity logic_and_voters is
  generic (
    NUM_TMR      : integer := 1);
  port (
    error_in      : in  std_logic;
    enable_1_n    : in  std_logic;
    enable_2_n    : in  std_logic;
    logic_in_1    : in  std_logic_vector(0 to NUM_TMR-1);
    logic_in_2    : in  std_logic_vector(0 to NUM_TMR-1);
    maj_voter_in_1 : in  std_logic_vector(0 to NUM_TMR-1);
    maj_voter_in_2 : in  std_logic_vector(0 to NUM_TMR-1);
    feedback_in   : in  std_logic;
    CLK           : in  std_logic;
    RST           : in  std_logic;
    majority_out  : out std_logic_vector(0 to NUM_TMR-1);
    minority_out  : out std_logic_vector(0 to NUM_TMR-1);
    logic_out     : out std_logic_vector(0 to NUM_TMR-1);
    voter_out    : out std_logic_vector(0 to NUM_TMR-1));
end logic_and_voters;

architecture Behavioral of logic_and_voters is

  component blink_module
    port (
      clk_in      : in  std_logic;
      rst         : in  std_logic;
      feedback_in : in  std_logic;
      error_in_n  : in  std_logic;
      blink_out   : out std_logic);
  end component;

  component majority_voter
    port (
      A : in  std_logic;
      B : in  std_logic;
      C : in  std_logic;
      Y : out std_logic);
  end component;

  component minority_voter
    port (
      P : in  std_logic;
      R1 : in std_logic;
      R2 : in std_logic;
      Y : out std_logic);
  end component;

  signal logic_signal      : std_logic_vector(0 to NUM_TMR-1);
  signal majority_out_signal : std_logic_vector(0 to NUM_TMR-1);
  signal minority_out_signal : std_logic_vector(0 to NUM_TMR-1);
  signal maj_voter_in_1_en : std_logic_vector(0 to NUM_TMR-1);
  signal maj_voter_in_2_en : std_logic_vector(0 to NUM_TMR-1);
  signal logic_in_1_en     : std_logic_vector(0 to NUM_TMR-1);
  signal logic_in_2_en     : std_logic_vector(0 to NUM_TMR-1);

begin

  -- disable the majority voter and logic values from the other modules if they
  -- are being reprogrammed
  maj_voter_in_1_en <= maj_voter_in_1 when enable_1_n = '0' else (others => '0');
  maj_voter_in_2_en <= maj_voter_in_2 when enable_2_n = '0' else (others => '0');

  logic_in_1_en <= logic_in_1 when enable_1_n = '0' else (others => '0');
  logic_in_2_en <= logic_in_2 when enable_2_n = '0' else (others => '0');

  -- instantiate NUM_TMR copies of the majority voting module
  majority_voters : for i in 0 to NUM_TMR-1 generate
    maj_voter      : majority_voter
      port map (

```

```

        A => logic_signal(i),
        B => logic_in_1_en(i),
        C => logic_in_2_en(i),
        Y => majority_out_signal(i));
end generate majority_voters;

-- instantiate NUM_TMR copies of the minority voting module
minority_voters : for i in 0 to NUM_TMR-1 generate
    min_voter    : minority_voter
        port map (
            P => majority_out_signal(i),
            R1 => maj_voter_in_1_en(i),
            R2 => maj_voter_in_2_en(i),
            Y => minority_out_signal(i));
end generate minority_voters;

majority_out <= majority_out_signal;
minority_out <= minority_out_signal;

-- logic module - LED-blinking circuitry that blinks an LED as 1Hz until an
-- error is inflicted, which causes it to blink at 8 Hz.
blink_module_1 : blink_module
    port map (
        clk_in      => CLK,
        rst         => RST,
        feedback_in => feedback_in,
        error_in_n  => error_in,
        blink_out   => logic_signal(0));

logic_out <= logic_signal;

-- only output the majority voter signal if it is not in the minority
voter_out <= majority_out_signal and not minority_out_signal;

end Behavioral;

```

## B.9: Reconfigurable Module – LED Blink – VHDL

```

-----
-- Worcester Polytechnic Institute
-- General Dynamics C4 Systems MQP
-- Evan Custodio
-- Brian Marsland
-- D Term 2007
-- Self-Healing Partial Reconfiguration of an FPGA
-- Filename: blink_module.vhd
-----

-- This module is the logic module used in the self-healing design. It blinks
-- an LED at 1 Hz until an "error" is asserted, at which point it begins
-- blinking the LED at 8 Hz until it is reset. It also includes synchronization
-- logic for the majority-voted feedback signal.
-----

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity blink_module is
    port (clk_in      : in  std_logic;
          rst         : in  std_logic;
          feedback_in : in  std_logic;
          error_in_n  : in  std_logic;
          blink_out   : out std_logic);

```

```

end blink_module;

architecture Behavioral of blink_module is
    signal blink_fast_signal : std_logic; -- 8 Hz blink
    signal blink_slow_signal : std_logic; -- 1 Hz blink
    signal error_in_hold    : std_logic;
    signal feedback_in_ld   : std_logic;
    signal error_in         : std_logic;
begin
    error_in <= not error_in_n;

    -- output the 88 Hz clock if an error has been inflicted, otherwise output
    -- the 1 Hz clock
    blink_out <= blink_fast_signal when (error_in='1' or error_in_hold='1') else
        blink_slow_signal;

    -- generate the 1 Hz and 8 Hz clock signals, as well as register the input
    -- error signal
    process(clk_in)
        variable counter_slow : integer range 0 to 50_000_000;
        variable counter_fast : integer range 0 to 6_250_000;
    begin
        if Clk_in'event and Clk_in = '1' then
            if rst = '1' then
                counter_slow := 0;
                counter_fast := 0;
                blink_slow_signal <= '0';
                blink_fast_signal <= '0';
                feedback_in_ld <= '0';
                error_in_hold <= '0';
            else
                -- synchronization logic: if the majority value switches states, reset
                -- the counters and set the blinking signal to the majority value
                if (feedback_in_ld /= feedback_in) then
                    counter_slow := 0;
                    counter_fast := 0;

                    blink_slow_signal <= feedback_in;
                    blink_fast_signal <= feedback_in;
                end if;

                counter_slow := counter_slow + 1;
                counter_fast := counter_fast + 1;

                if counter_slow = 50_000_000 then
                    blink_slow_signal <= not blink_slow_signal;
                    counter_slow := 0;
                end if;
                if counter_fast = 6_250_000 then
                    blink_fast_signal <= not blink_fast_signal;
                    counter_fast := 0;
                end if;

                -- register the feedback for one clock cycle for comparison purposes
                feedback_in_ld <= feedback_in;

                -- latch any input errors
                if error_in = '1' then
                    error_in_hold <= '1';
                end if;
            end if;
        end if;
    end process;
end Behavioral;

```

## B.10: Reconfigurable Module – Majority Voter – VHDL

```
-----  
-----  
-- Worcester Polytechnic Institute  
-- General Dynamics C4 Systems MQP  
-- Evan Custodio  
-- Brian Marsland  
-- D Term 2007  
-- Self-Healing Partial Reconfiguration of an FPGA  
-- Filename: majority_voter.vhd  
-----  
-----  
-- This module is the majority voter used in the TMR design.  
-----  
-----  
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.STD_LOGIC_ARITH.all;  
use IEEE.STD_LOGIC_UNSIGNED.all;  
  
entity majority_voter is  
  port ( A : in  std_logic;  
         B : in  std_logic;  
         C : in  std_logic;  
         Y : out std_logic);  
end majority_voter;  
  
architecture Behavioral of majority_voter is  
begin  
  
  Y <= (A and B) or (A and C) or (B and C);  
  
end Behavioral;
```

## B.11: Reconfigurable Module – Minority Voter – VHDL

```
-----  
-----  
-- Worcester Polytechnic Institute  
-- General Dynamics C4 Systems MQP  
-- Evan Custodio  
-- Brian Marsland  
-- D Term 2007  
-- Self-Healing Partial Reconfiguration of an FPGA  
-- Filename: minority_voter.vhd  
-----  
-----  
-- This module is the minority voter used in the TMR design.  
-----  
-----  
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.STD_LOGIC_ARITH.all;  
use IEEE.STD_LOGIC_UNSIGNED.all;  
  
---- Uncomment the following library declaration if instantiating  
---- any Xilinx primitives in this code.  
--library UNISIM;  
--use UNISIM.VComponents.all;  
  
entity minority_voter is  
  port ( P : in  std_logic;  
         R1 : in  std_logic;
```

```

        R2 : in  std_logic;
        Y  : out std_logic);
end minority_voter;

architecture Behavioral of minority_voter is

begin

    Y <= ((not P) and R1 and R2) or (P and (not R1) and (not R2));

end Behavioral;

```

## B.12: Static Module – Safe Module – VHDL

```

-----
-----
-- Worcester Polytechnic Institute
-- General Dynamics C4 Systems MQP
-- Evan Custodio
-- Brian Marsland
-- D Term 2007
-- Self-Healing Partial Reconfiguration of an FPGA
-- Filename: safe_module.vhd
-----
-----
-- This module is the safe module used in the TMR design.
-----
-----

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity safe_module is
    generic (
        NUM_TMR : integer := 1);
    port (
        voter_1 : in  std_logic_vector(0 to NUM_TMR-1);
        voter_2 : in  std_logic_vector(0 to NUM_TMR-1);
        voter_3 : in  std_logic_vector(0 to NUM_TMR-1);
        enable_n : in  std_logic_vector(0 to 2);
        X       : out std_logic_vector(0 to NUM_TMR-1));
end safe_module;

architecture Behavioral of safe_module is
    signal voter_1_en : std_logic_vector(0 to NUM_TMR-1);
    signal voter_2_en : std_logic_vector(0 to NUM_TMR-1);
    signal voter_3_en : std_logic_vector(0 to NUM_TMR-1);
begin

    voter_1_en <= voter_1 when enable_n(0) = '0' else (others => '0');
    voter_2_en <= voter_2 when enable_n(1) = '0' else (others => '0');
    voter_3_en <= voter_3 when enable_n(2) = '0' else (others => '0');

    X <= voter_1_en or voter_2_en or voter_3_en;

end Behavioral;

```

## B.13: User Constraints File

```
NET "CLK" LOC = AH17;
NET "RST" LOC = G18;
NET "error_in<0>" LOC = H25;
NET "error_in<1>" LOC = G26;
NET "error_in<2>" LOC = H26;
NET "tmr2ppc_logic<0>" LOC = E31;
NET "tmr2ppc_logic<1>" LOC = E32;
NET "tmr2ppc_logic<2>" LOC = F31;
NET "tmr2ppc_min_voters<0>" LOC = E1;
NET "tmr2ppc_min_voters<1>" LOC = E2;
NET "tmr2ppc_min_voters<2>" LOC = E3;
NET "X" LOC = E4;

#####
## This system.ucf file is generated by Base System Builder based on the
## settings in the selected Xilinx Board Definition file. Please add other
## user constraints to this file based on customer design specifications.
#####

#Net sys_clk_pin LOC=AH17;
#Net sys_rst_pin LOC=G25;
#Net sys_rst_pin PULLUP;
## System level constraints
#Net sys_clk_pin TNM_NET = sys_clk_pin;
#TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 10000 ps;
#Net sys_rst_pin TIG;

#NET "RSTC405RESETSYS" TPTHU = "RST_GRP";
#NET "RSTC405RESETCCHIP" TPTHU = "RST_GRP";
#NET "RSTC405RESETCORE" TPTHU = "RST_GRP";
#NET "C405RSTCHIPRESETRREQ" TPTHU = "RST_GRP";
#NET "C405RSTCORERESETRREQ" TPTHU = "RST_GRP";
#NET "C405RSTSYSRESETRREQ" TPTHU = "RST_GRP";

#TIMESPEC "TS_RST1" = FROM CPUS THRU RST_GRP TO FFS TIG;
#TIMESPEC "TS_RST2" = FROM FFS THRU RST_GRP TO FFS TIG;
#TIMESPEC "TS_RST3" = FROM FFS THRU RST_GRP TO CPUS TIG;

## FPGA pin constraints
Net fpga_0_RS232_1_RX_pin LOC=AJ19;
Net fpga_0_RS232_1_TX_pin LOC=AK19;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<29> LOC=K16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<28> LOC=AD17;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<27> LOC=AE17;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<26> LOC=K14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<25> LOC=E6;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<24> LOC=AE16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<23> LOC=AE15;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<22> LOC=AE14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<21> LOC=D5;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<20> LOC=D9;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<19> LOC=E16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<18> LOC=F17;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<17> LOC=C9;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<16> LOC=D10;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<15> LOC=D12;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<14> LOC=F16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<13> LOC=F15;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<12> LOC=K15;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<11> LOC=K12;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<10> LOC=D6;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_A_pin<9> LOC=K11;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<31> LOC=G10;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<30> LOC=H14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<29> LOC=H10;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<28> LOC=F13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<27> LOC=D15;
```



```

Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<26> LOC=D16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<25> LOC=D13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<24> LOC=D14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<23> LOC=E10;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<22> LOC=G9;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<21> LOC=E13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<20> LOC=H9;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<19> LOC=C14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<18> LOC=F14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<17> LOC=E14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<16> LOC=C13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<15> LOC=K17;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<14> LOC=J16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<13> LOC=L17;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<12> LOC=J10;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<11> LOC=L16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<10> LOC=G17;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<9> LOC=G14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<8> LOC=G15;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<7> LOC=J14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<6> LOC=J15;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<5> LOC=J12;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<4> LOC=J11;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<3> LOC=G16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<2> LOC=H16;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<1> LOC=E7;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_DQ_pin<0> LOC=E9;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin<3> LOC=AF11;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin<2> LOC=AF13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin<1> LOC=AJ13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_BEN_pin<0> LOC=AK14;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_WEN_pin LOC=AE11;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_OEN_pin<0> LOC=H13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_OEN_pin<1> LOC=C11;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_CEN_pin<0> LOC=G13;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_CEN_pin<1> LOC=F9;
Net fpga_0_SRAM_256Kx32_FLASH_1Mx32_Mem_RPN_pin LOC=AL11;

Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<31> LOC=AB3;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<30> LOC=V4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<29> LOC=AB4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<28> LOC=W3;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<27> LOC=AA4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<26> LOC=W4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<25> LOC=AA3;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<24> LOC=Y4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<23> LOC=Y1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<22> LOC=AA1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<21> LOC=Y2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<20> LOC=AB1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<19> LOC=AA2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<18> LOC=AC1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<17> LOC=AB2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<16> LOC=AC2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<15> LOC=AE4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<14> LOC=AF4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<13> LOC=AF3;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<12> LOC=AK4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<11> LOC=AK3;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<10> LOC=AC4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<9> LOC=AC3;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<8> LOC=AD4;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<7> LOC=AD2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<6> LOC=AE2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<5> LOC=AE1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<4> LOC=AG1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<3> LOC=AF2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<2> LOC=AL1;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<1> LOC=AG2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_DQ_pin<0> LOC=AL2;
Net fpga_0_SDRAM_8Mx32_1_SDRAM_Addr_pin<11> LOC=AD6;

```

Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_Addr\_pin<10> LOC=W5;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_Addr\_pin<9> LOC=V5;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_Addr\_pin<8> LOC=AH6;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_Addr\_pin<7> LOC=Y6;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_Addr\_pin<6> LOC=V7;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_Addr\_pin<5> LOC=W6;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_Addr\_pin<4> LOC=W7;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_Addr\_pin<3> LOC=Y7;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_Addr\_pin<2> LOC=AA6;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_Addr\_pin<1> LOC=AA5;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_Addr\_pin<0> LOC=AB7;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_DQM\_pin<3> LOC=Y3;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_DQM\_pin<2> LOC=W2;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_DQM\_pin<1> LOC=AD3;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_DQM\_pin<0> LOC=AD1;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_WEn\_pin LOC=AE5;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_CKE\_pin LOC=AB6;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_CS\_n\_pin LOC=AB5;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_CAS\_n\_pin LOC=AC6;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_RAS\_n\_pin LOC=AH5;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_Clk\_pin LOC=AC7;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_BankAddr\_pin<1> LOC=V6;  
Net fpga\_0\_SDRAM\_8Mx32\_1\_SDRAM\_BankAddr\_pin<0> LOC=AD5;

## Appendix C: Project Goals

- Implement a cryptographic circuit that includes triple redundancy for error checking and the capability of healing itself if an error is detected.
  - Design a self-reconfiguring system
    - Re-implement the previous project group's design. (1)
    - Familiarize with PowerPC and Xilinx Platform Studio. (2)
    - Prove functionality of the PowerPC in a partially reconfigurable design. (3)
    - Achieve communication with the ICAP port.
      - Use the IP Core of the PowerPC. (3)
      - Or, design a custom VHDL solution. (5)
    - Implement a simple ICAP design into a partially reconfigurable environment. (4)
    - Load a partial bitstream from Flash memory through the ICAP port to the partial area of the FPGA on button push. (5)
  - Design a self-healing system
    - Implement TMR functionality to detect errors in the redundant modules.
      - Use custom VHDL to replicate modules and create voting circuitry. (4)
      - Or, use the Xilinx tool to implement TMR. (5)
    - Modify the ICAP design to load a partial bitstream to a redundant module when an error in that module is detected. (5)
  - Design a more complex VHDL module in the self-healing system.
    - Use a supplied AES module if it can fit within the size constraints. (2)
    - Or, create and use an alternate custom algorithm. (2)

Note: The number following each lowest-level goal is the estimated difficulty on a scale of 1 (easiest) to 5 (hardest).