

Dynamic Partial Reconfiguration of a Field Programmable Gate Array

A Major Qualifying Project Report
Submitted to the faculty of
Worcester Polytechnic Institute
In partial fulfillment of the requirements for the
Degree of Bachelor of Science

Submitted By:

Michael Kristan

Brian Loveland

Robert Sazanowicz

Sponsored by:

General Dynamics C4 Systems
77 A Street
Needham, MA 02494

Liaisons :

Brendon Chetwynd
Orlando Gerardo

Submitted To:

Prof. John McNeill

Co-Advised By: Prof. Berk Sunar



GENERAL DYNAMICS
C4 Systems

Abstract

The goal of this project was to develop a prototype of real-time partial re-configuration of a logic circuit. The steps required for completion involved researching possible circuit algorithms, implementing desired functionality in VHDL, and developing a proof of concept. The objective of this project was to lay a foundation for further development in implementing a self-healing triple modular redundant AES encryption system. The project was completed under the guidance of Brendon Chetwynd and Gerardo Orlando of General Dynamics C4 Systems.

Authorship

The completion of this Major Qualifying Project involved a significant amount of time and dedication from each respective team member. This report represents a collaboration of ideas from all members as well as an equal level of time and effort. Throughout the course of this project, several aspects of work were divided among team members. Rob Sazanowicz focused his work on PlanAhead development and the implementation of the Partial Reconfiguration over JTAG designs. Brian Loveland focused his efforts on the implementation of self-reconfiguration through use of the Embedded Development Kit and the on board PowerPC processors. Mike Kristan contributed greatly to the project management aspect as well as working on the integration of Embedded Development Kit and PlanAhead tool flows with Rob. The separation of these tasks allowed us to specialize our work to increase productivity. However, much of the project's success was dependant on peer collaboration. Each team member contributed equally to this project report in all aspects including research, practice, and writing.

Executive Summary

General Dynamics C4 Systems implements several technologies in their development of hardware encryption systems as a US government contractor. One such technology involves the development of encryption systems through the use of a Field Programmable Gate Array (FPGA). First developed in 1984, FPGAs represent a growing technology of which functionality and ease of use are constantly improving. One such development involves dynamic partial reconfiguration, the ability to reprogram a portion of the FPGA without powering down or resetting the chip. This functionality is still considered to be academic in nature and has seen little professional development. This ability opens the door to new applications as well the reduction of size requirements for designs. This is concluded from the idea that larger designs can be broken into smaller, less resource intensive partial designs and then programmed to the chip only when necessary. In addition, with reconfigurable logic such as an FPGA, there is a possibility that it could inadvertently loose or change its configuration. Partial reconfiguration, tied with the necessary circuitry, would allow the FPGA to identify these issues and reconfigure itself only where the faults occur. Xilinx, the original developers of the FPGA, have pushed development of partially reconfigurable designs.

The Xilinx Virtex II Pro series FPGAs allow for support of partial reconfiguration through the use of an enhanced toolset. The primary objective of this Major Qualifying Project was a proof of concept for a partially reconfigurable design. To accomplish this, the team began development on a Memec Design XC2VP30 – FFII152 development board. The on board Virtex II Pro FPGA supports dynamic partial reconfiguration and contains an embedded IBM PowerPC processor to control self-reconfiguration. The secondary objective of this project is to implement a self-healing design through the use of one of these embedded processors. The PowerPC has the ability to control reconfiguration of the FPGA through the Internal Configuration Access Port (ICAP). Ideally, a PR design would be able to detect a need for and control partial reconfiguration creating a self-sustaining system. Eliminating the need for an external processor lowers costs and size requirements while simplifying the overall design and increasing reliability by not relying on external interconnects. The third major goal of this project is the creation of a base for future development with partial reconfiguration. One such development includes the integration of a Triple Modular Redundant (TMR) system with partially reconfigurable capabilities. TMR has been widely used to increase the reliability of FPGA circuits and is used to avoid corruption due to environment hazards and side channel attacks.

In order to complete our primary objective, the development of a proof of concept for partial reconfiguration, it was first necessary to gather the required software and hardware tools and create a design flow. A handful of third party tutorials as well as Xilinx provided documentation was used to develop a design flow which functioned properly with the toolset available to us. The partial reconfiguration design flow follows closely the

modular design flow most commonly used by industry professionals with the addition and modification of a few software tools. Once the necessary research was completed to collect and learn these tools, the team began creation of an initial PR design. This basic circuit was programmed through the use of an external PC over the JTAG connection. Functionality of the design was simple; a single reconfigurable module which could be implemented as either an addition or subtraction unit and a single static module which controlled the function of 4 on board light emitting diodes (LEDs) were written in VHDL and synthesized following the modular design flow. The design was created such that proof of reconfiguration could be tested and monitored with the use of an oscilloscope. The team was successful in the development of a proof of concept for partial reconfiguration.

Development of a self-reconfiguring system, the secondary objective of this project, proved to be a much more complex task. Similarly to the primary objective, development of this system required the use of a new design flow and toolset. The integration of PowerPC peripherals to the design required the integration of two separate design methodologies. The PowerPC design (using the Xilinx Embedded Development Kit) would need to be merged with the previous PR design (using Xilinx PlanAhead) to create a self reconfiguring system. Although the completion of a self-reconfiguring system was not realized, several milestones were achieved during development. The team was able to demonstrate the ability to program and load a PowerPC design to the development board. Also, the ability to write and read from an external flash component was demonstrated. This is required because a significant amount of memory is needed to store the configuration files handled by the PowerPC. Unfortunately, the integration of a PowerPC design and the previous PR design was not completed as many complications prevented successful combination of the two design flows. The team is confident however that with more time and the development of a more integrated tool flow, this design could be accomplished.

The third and final objective of this project was to create a project base for which future students can branch off of. This included the use of a Version Object Base (VOB), using the Rational ClearCase tool, which all research and work material was stored on. The VOB for this project was initially created by General Dynamics and will be made available for any future project to view and work from. In order to allow for a smooth transition to the subsequent project, the team needed to setup a directory structure as well as make significant documentation of steps taken throughout the process. The development of the first two objectives was monitored and noted throughout the project so the following project(s) could minimize transition time. With the addition of code and project notes, useful reading and tutorials used were provided as well as research examples which were found to be helpful in understanding the Partial Reconfiguration design process.

Overall, the completion of these objectives proved to be taxing as the software available did not always function as promised. Partial Reconfiguration is still a fairly underused practice and the tools for creating these types of designs have not yet caught up with the hardware's full capabilities. However, the primary goal, a proof of concept for Dynamic Partial Reconfiguration was realized. In doing so, a basic foundation for future projects to branch from was created, opening the doors for further advances in partial reconfiguration design.

Acknowledgements

We would like to thank General Dynamics C4 Systems as well as Worcester Polytechnic Institute for providing us with the opportunity to complete this Major Qualifying Project. Furthermore, we would like to thank Brendon Chetwynd for his constant support on site at General Dynamics. Brendon proved to be a constant motivator and never failed to supply us with a sometimes never-ending request for software and hardware tools. We would also like to thank Gerardo Orlando for his technical advice and support throughout the course of the project. We also thank Chris Chalifoux of General Dynamics and Natalie Chin (WPI '01) for providing ClearCase assistance and technical support.

Finally, we would like to thank Professor John McNeill for advising this project and keeping this opportunity open to us in the absence of Professor Berk Sunar. This project would not have been completed without his guidance during the preparation phase or willingness to take on an advising role normally outside his scope of interest.

Table of Contents

Abstract	ii
Authorship.....	iii
Executive Summary	iv
Acknowledgements	vii
Table of Contents.....	viii
Table of Figures.....	xii
1: Introduction.....	1
2: Background	3
2.1: Field Programmable Gate Arrays (FPGA)	3
2.2: Partial Reconfiguration (PR).....	3
2.2.1: Types of partial reconfiguration	4
2.2.2: PR example and benefits	4
2.3: Available Tools	4
2.3.1: Memec Virtex II – Pro FF1152 Development Board	4
2.3.2: Xilinx ISE 8.2i.....	5
2.3.3: Xilinx iMPACT.....	6
2.3.4: Xilinx PlanAhead 8.2.7.....	6
2.3.5: Xilinx Embedded Development Kit (EDK).....	7
2.3.6: Rational ClearCase	8
2.4: Previous Work.....	10
2.5 : General Dynamics C4 Systems (GDC4S).....	10
2.6: Advanced Encryption Standard (AES)	11
2.7: Triple Modular Redundancy (TMR).....	12
2.8: TMR, AES, and PR Together	13
3: Design Requirements.....	15
3.1: Modular Design Flow	15
3.2: Modular Partial Reconfiguration Flow.....	16
3.3: Test Circuit Overview.....	16
3.3.1: Preliminary Requirements.....	16
3.3.2: Programming Methods.....	17
3.4: Process Overview	18
3.4.1: Design and Synthesis	18
3.4.2: Budgeting and Constraints.....	19

3.4.3: Implementation and Assembly	19
3.4.4: Download and Testing	20
4: Implementation – Basic Circuit	21
4.1: Software Requirements.....	21
4.2: HDL Design and Synthesis	21
4.2.1: Design Overview.....	21
4.2.2 Design Intent.....	22
4.2.3: Top Level	22
4.2.4: Module Level.....	23
4.2.5: Bus Macros.....	24
4.3: Implementation with PlanAhead	24
4.3.1: Project Creation.....	24
4.3.2: Budgeting and Constraints.....	25
4.3.3: Partial Re-config Tool	27
4.4: Programming.....	29
4.4.1: Configuration over JTAG.....	29
4.5: Creation of a PR AES Implementation.....	29
4.5.1: Design Overview	29
4.5.2: Limitations.....	30
4.5.3: Steps Required.....	30
4.5.4: Feasibility	30
4.6: Testing and Results – Basic Circuit	31
4.6.1: Overview.....	31
4.6.2: Bit Stream Differences	31
4.6.3: Desired Functionality – Configuration over JTAG	32
4.6.4: Results – Configuration over JTAG.....	32
5: Self-Partial Reconfiguration Implementation.....	35
5.1: Overview.....	35
5.1.1: Advantages over JTAG Partial Reconfiguration	35
5.1.2: PowerPC vs. Custom Logic.....	35
5.2: Design Procedure.....	36
5.2.1: Overview	36
5.2.2: BRAM vs. Flash Memory	37
5.2.3: Programming Flash Memory	37
5.2.4: Programming - Configuration over ICAP / PowerPC	39

5.2.5: PlanAhead Development	40
5.2.6: Flow Integration	41
5.3: Issues	42
5.3.1: Lack of design flow.....	42
5.3.2: Removal of DCM wrappers.....	42
5.3.3: EDK Makefiles	43
5.3.4: Synthesis parameters	43
5.3.5: AREA_GROUP errors when mapping bus macros.....	43
5.3.6: Memory mapping errors on static implementation.....	43
5.4.7: ClearCase check-in errors for binary files.....	44
5.4: Results	44
5.4.1: Successes	44
5.4.2: Failures	45
Conclusion	46
Goal Completion	46
Future Work	47
References.....	48
Glossary	50
Appendix A - Basic Circuit – PR over JTAG Design.....	54
A.1: Top Level - VHDL	54
A.2: Reconfigurable Module – Addition Logic - VHDL.....	56
A.3: Reconfigurable Module – Subtraction Logic - VHDL	56
A.4: Static Module – LED Display – VHDL	57
A.5: Clock Converter Module	58
Appendix B – Self Reconfiguration – PR over ICAP design.....	60
B.1: XST synthesis parameter file (system_xst.scr).....	60
B.2: Sample ICAP test program	60
B.3: HWICAP User-defined tools – Header file	61
B.4: HWICAP User-defined tools – Implementation code	62
B.5: Bit file to array Perl tool	63
B.7: Bitstream Combiner Script.....	65
B.8: Diff on customized system.vhd based on generated file.....	65
Appendix C: Basic PR – Troubleshooting Guide.....	69
Appendix D: EDK Troubleshooting Guide	72
Appendix E: ClearCase information	73

E.1: VOB directory structure	73
E.2: ClearCase config spec	73
E.3: Add to source control recursive script (add-to-src-control.bat)	74
E.4: Timestamp label generator (datelabel.sh).....	76
E.5: Miscellaneous recursive commands	76
E.5.1: Revert checked out files to the VOB version (revert.bat).....	76
E.5.2: Push changes to VOB but keep elements checked out (push.bat).....	77
E.5.3: Recursive unreserved check out	77
E.5.4: Recursive check in	77
E.6: ClearCase Menu Administrator	77
E.7: Snapshot views vs. Dynamic views	78

Table of Figures

Figure 2.1 – Memec Virtex II Pro ff1152 Dev. Board	5
Figure 2.2 – ISE Design Environment	5
Figure 2.3 – Xilinx Impact Software	6
Figure 2.4 – PlanAhead Design Environment	7
Figure 2.5 – Xilinx EDK Design Environment.....	8
Figure 2.6 - Version tree of an element	9
Figure 2.7 - Diff viewer showing changes between versions.....	10
Figure 2.9 – Triple Modular Redundancy Description.....	12
Figure 2.10 – Triple Modular Redundant System Description	12
Figure 3.1 – Modular Design Flow	15
Figure 4.1 – Top Level Design Hierarchy	21
Figure 4.2 – XST Process Properties.....	23
Figure 4.3 – Reconfigurable Module (Yellow) in PlanAhead Device View.....	25
Figure 4.4 – Bus Macro Placement (Various Colors).....	26
Figure 4.5 – Partial Reconfig GUI	28
Figure 4.6 – 2Hz LED	32
Figure 4.7 – 20KHz LED.....	32
Figure 4.8 – Solid LED in Off State	33
Figure 4.9 – Solid LED in ON State	33
Figure 4.10 – 20Khz LED w/ Reconfiguration.....	34
Figure 4.11 – Solid ON LED w/ Reconfiguration	34
Figure 5.1 – Top Level Design Hierarchy	37
Figure 5.2 – Flash Memory Programmer	38
Figure 5.3 – Flash Combination Script	39
Figure 5.4 – Xilinx SDK Screen Shot	40
Figure 5.5 - Map error on Static Implementation.....	44
Figure 5.6 - PowerPC is unreachable via JTAG and the Xilinx debugger.....	45

1: Introduction

As the protection of digital data becomes more important to society, reliable encryption systems are needed in order to properly safeguard confidential information. It has been shown that most failures of modern encryption systems are due to implementation failures, rather than algorithm failures, and therefore it is critical to ensure that the implementation is not compromised. Implementing an encryption system on reconfigurable logic such as a field programmable gate array (FPGA) is desirable for its ability to be easily reprogrammed like a microcontroller, but allows for the performance advantages of custom digital hardware like an application specific integrated circuit (ASIC). There is also a significantly lower cost for custom digital hardware of an FPGA when prototyping and for production in small quantities compared to an ASIC. The major disadvantage of FPGA devices is that they may be left vulnerable to side channel attacks or even accidental configuration changes, due to the fact that the configuration of the digital hardware can be altered. Fortunately, when the configuration is accidentally or maliciously changed, it is often the case that only a portion of the design becomes unusable. Therefore, in cooperation with General Dynamics C4 Systems, this project investigates designing a system which uses the partial reconfiguration feature of Xilinx FPGAs to repair or update portions of a circuit without disrupting normal operating activity.

Advances in FPGA hardware and software design tools for the FPGA hardware allow for on the fly reconfiguration of programmable logic. This ability allows programmers to design a chip which can be reconfigured without disrupting operation and can be used to recover a compromised device. Partial reconfiguration can also be used to allow larger complex designs to be implemented on devices with fewer resources than would normally be required for a complete implementation. The most common method for implementation of partial reconfiguration is one in which a modular design approach is used. The logic is configured in a manner such that one module is dynamically reconfigured while another module is retained in a static configuration for performing operations which do not change. With this approach, a portion of the chip is continuously being reconfigured to its original condition or to a new configuration regardless if an error is detected. Partial reconfiguration may also be implemented in conjunction with Triple Mode Redundancy (TMR) to create a system with increased reliability. TMR utilizes a voting circuit to detect differences in output from three identical systems. Often, this voting system is replicated three times as well, to further increase dependability. The ultimate goal is to develop a TMR system which utilizes partial reconfiguration to repair damaged modules in the background, offering seamless, reliable operation of a secure system.

The first step in the creation of a secure, redundant, self-reconfiguring system is the development of the partial reconfiguration module. In order to implement a partially reconfigurable encryption algorithm, a proof of concept of partial reconfiguration must be completed using a smaller, simpler system. The simple test system

includes the design of an arithmetic logic unit which allows the user to switch between operations causing the operation module to be reconfigured. This setup can be tested and easily interfaced with the available I/O ports on the development board. Upon successful completion of a partially reconfigurable circuit, the goal is to implement an advanced encryption algorithm with partially reconfigurable characteristics. Successful completion of this task will allow future project teams to integrate this algorithm with a TMR system.

2: Background

This project investigates the use of several technologies, some of which are still considered fairly academic and therefore are not standard digital logic nomenclature (specifically Partial Reconfiguration). In addition, there are some related technologies, which will not be implemented in our project due to time constraints, however they will be implemented in future projects based on our project, including TMR and AES.

2.1: Field Programmable Gate Arrays (FPGA)

FPGAs were invented in 1984 by Ross Freeman, co-founder of Xilinx [24]. FPGAs provide users with an environment where the user is able to quickly develop and implement a circuit or module at low cost and fast turnaround time. FPGAs are developed using simple circuitry that is meshed together to provide the same complex circuitry that a traditional circuit board provides. The drawback to using an FPGA is that unlike traditional application specific integrated circuits (ASICs), FPGA circuits have a high propagation delay and therefore a slower processing time [11].

Despite the drawbacks to using an FPGA over a traditional circuit board, FPGAs can be used quite easily with Triple Modular Redundancy circuits. Quick reprogramming and scrubbing techniques provide the end user with a reliable output that allows the circuit to be automatically reprogrammed without any end user maintenance.

2.2: Partial Reconfiguration (PR)

Partial reconfiguration is the practice of reprogramming only a portion of an FPGA. More specifically, dynamic partial reconfiguration denotes the ability to reprogram a portion of a circuit while it is operating. This is done without a need for the chip to power down or be reset. For the sake of brevity, dynamic partial reconfiguration will be considered the same as partial reconfiguration in the remainder of this report. Partial reconfiguration can be accomplished with the aid of an external device such as a JTAG connector or can be managed by an on-board processor. Partial reconfiguration can be implemented through the use of the Xilinx ISE tools either stand alone or in conjunction with PlanAhead software. Dependent on design intent, partial reconfiguration generally requires the use of a modular design flow. Modular design requires additional procedures for synthesis and implementation and adds complexity not found when utilizing the basic design flow. In general, modular design allows for simultaneous development of different modules which together complete the design of an FPGA. This design procedure is most commonly used by engineering and programming teams who must coordinate efforts to save time and money. Modular design also allows a team to modify non-working or unstable modules without affecting those which are functioning properly. The

modular design flow consists of two phases. Phase 1 incorporates Design Entry and Synthesis in which the team leader completes a top level design and each team member completes the design entry and synthesis for their particular module. Phase 2 incorporates the creation of top level constraints, implementation of each module, and final assembly.

2.2.1: Types of partial reconfiguration

There are two types of partial reconfiguration, modular and difference-based. These design procedures hold certain advantages to each other, and each is limited in ability. Modular partial reconfiguration allows for portions of the FPGA to be recreated or even redesigned. This is most commonly used to implement large designs on smaller and less expensive chips. This is generally accomplished by splitting the design and reconfiguring one module over another in order to save resources. Modular reconfiguration can also be used to repair “broken” chip modules without affecting the operation of other areas of the chip when seamless operation is crucial to the application. Difference-based partial reconfiguration follows a different design flow. This approach requires the creation of a bit stream file which only includes design differences from one design to another. The initial design is created and then the Xilinx FPGA editor can be used to design logic changes directly on the FPGA. A partial bit stream file is then created which when loaded only makes changes based on the difference in the two designs. This flow allows for extremely quick transitions between the two designs but is limited in scope to what it can be used for.

2.2.2: PR example and benefits

An example of a system that could benefit from partial reconfiguration would be a basic arithmetic logic unit (ALU). The idea would be to make the math and logic operations modular in design so that depending on the function that is needed, the appropriate module could be inserted into the FPGA. Although this process is essentially unnecessary for such a simple circuit, in the case of larger and more demanding circuits a smaller FPGA could be used as opposed to one that could store all logic circuits at the same time. Creation of a simple ALU is the first process for a proof of concept of partial reconfiguration and will lay the groundwork for development of a more advanced encryption algorithm with partial reconfigurability.

2.3: Available Tools

2.3.1: Memec Virtex II – Pro FF1152 Development Board

The Memec Development board used for this project includes a Xilinx XCVP30 FPGA. This board is designed for extreme flexibility in high end applications. The development board includes two 32mb SDRAM blocks as well

as 2 integrated PowerPC processors. Xilinx 8.2i ISE software allows for programming through either Parallel or USB to JTAG connections. [14]

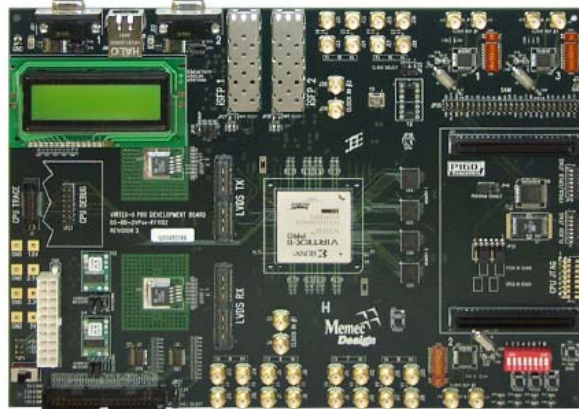


Figure 2.1 – Memec Virtex II Pro ff1152 Dev. Board

2.3.2: Xilinx ISE 8.2i

The primary tool used to design circuits is the Xilinx ISE. ISE allows users to design circuits in both schematics and VHDL [26]. Basic simulation can be done in Xilinx before it is sent to the board. Xilinx provides a number of tutorials and a large number of online resources that serve as an excellent reference.

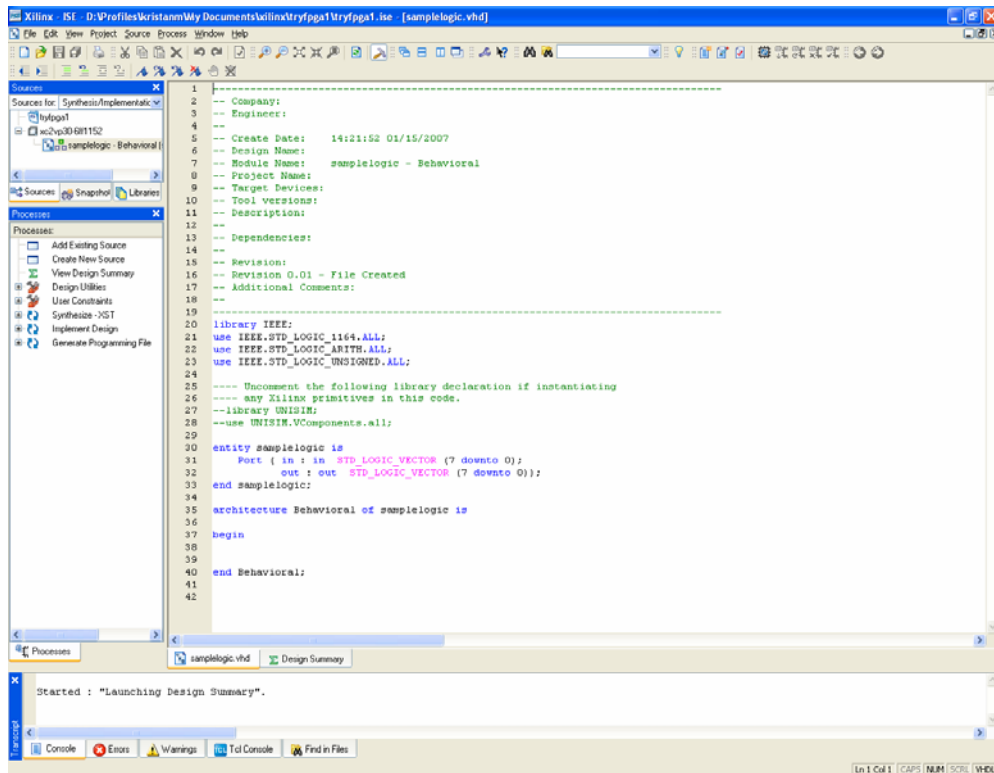


Figure 2.2 – ISE Design Environment

2.3.3: Xilinx iMPACT

iMPACT is the tool that is used to program the FPGA [25]. iMPACT can be used in a direct SPI configuration or in a boundary scan configuration. iMPACT also provides the user with the ability to erase, program, and verify the FPGA itself or the PROM on the board. The advantage to programming the PROM is that changes are retained after the device shuts off so that there is no need to reprogram the device after hitting the reset button or by turning on the power. Aside from providing options for downloading to the Xilinx device, iMPACT also provides the capability of merging bit streams with block memory maps provided by the EDK development.

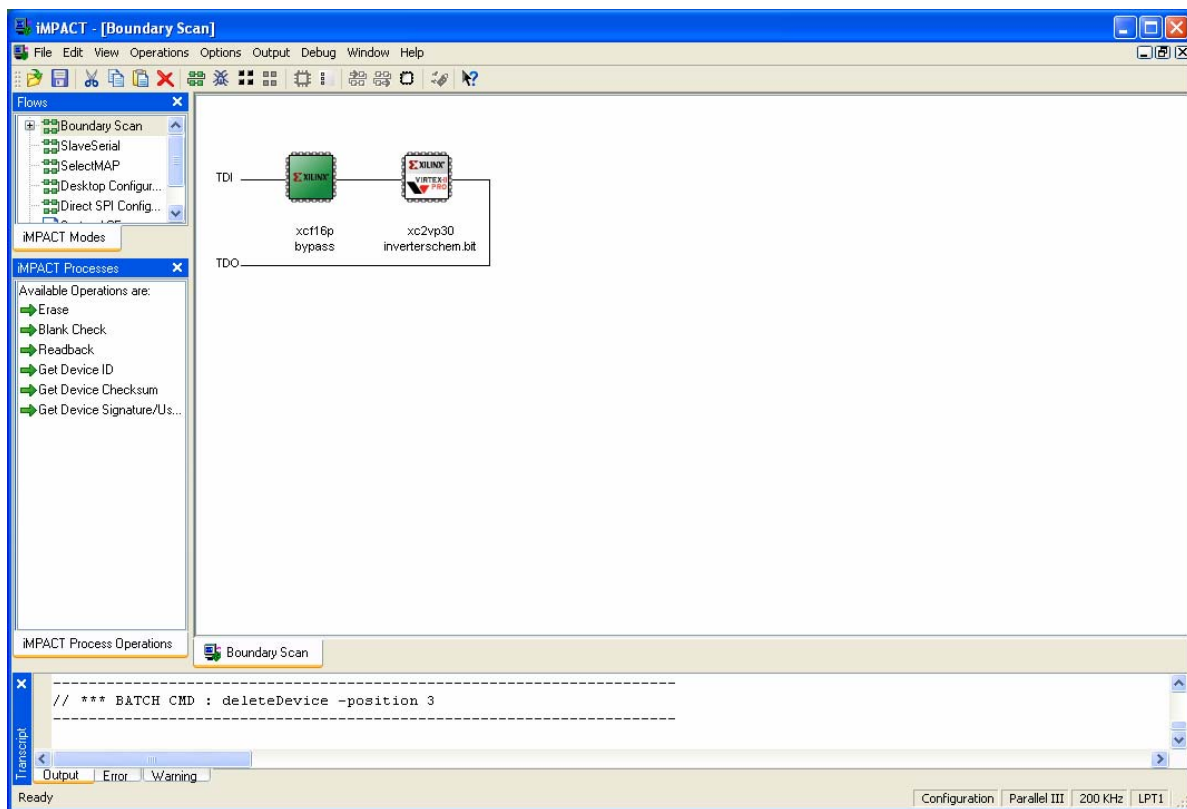


Figure 2.3 – Xilinx Impact Software

2.3.4: Xilinx PlanAhead 8.2.7

PlanAhead is another component of the Xilinx Design Studio. PlanAhead's main purpose is to customize the way circuits designed in ISE are laid out on the FPGA as well as providing timing and placement analysis to improve circuit function [19]. By using this tool, users can group circuits and modules. The benefit of this is that if each module is in its own area, the task of partial reconfiguration becomes much easier as only one area is being programmed and will not affect any of the other sections. This task is known as floor planning. Separation of modules and implementation of bus macros is facilitated within the software as the numerous

tools required to complete these procedures are integrated into a single platform. These tasks can be completed in Xilinx ISE tools however it is more difficult to coordinate these changes as scripting may be required. PlanAhead also provides a useful set of design rule checks which can be used to improve designs and provide suggestions to the designer. The integration of several features into a single piece of software allows the designer to use a simplified set of tools which are better designed to work with each other. DRC allows for error checking to save time further in the design flow.

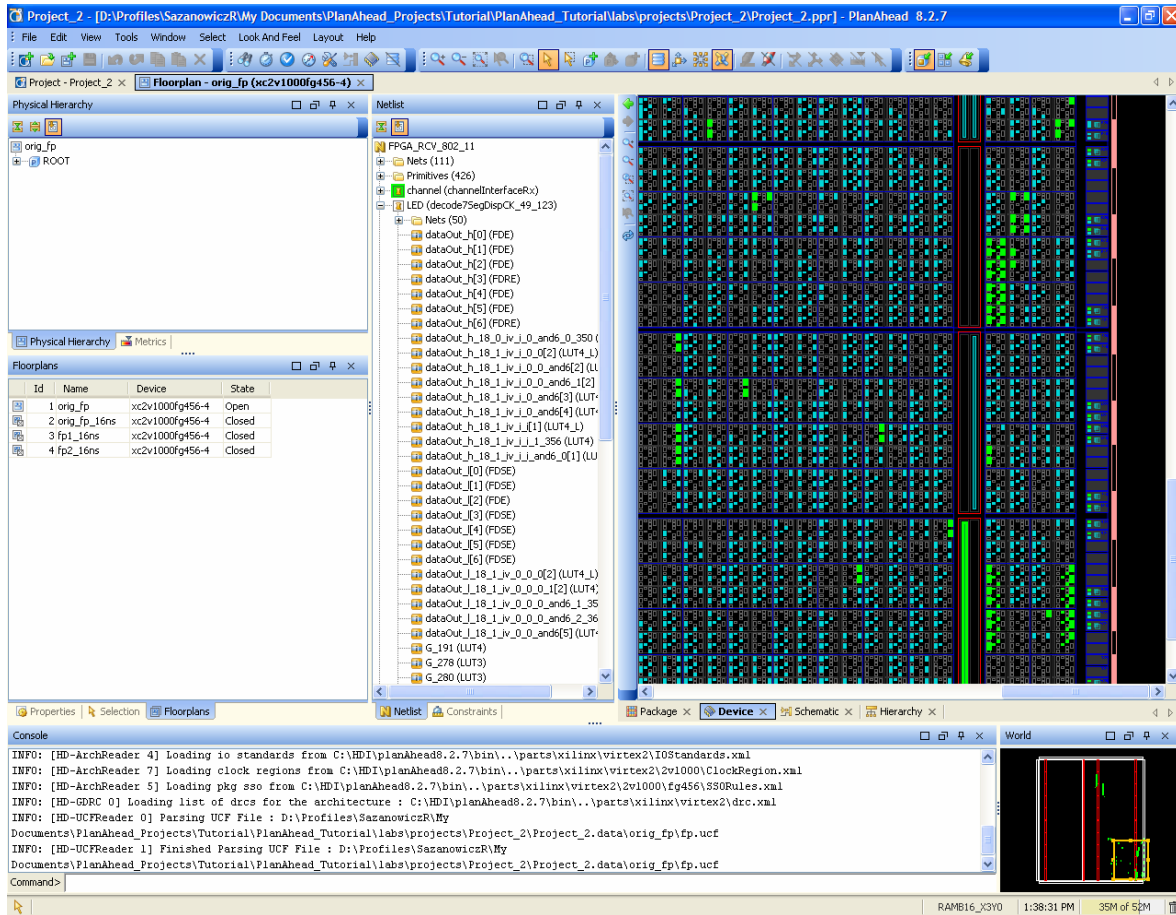


Figure 2.4 – PlanAhead Design Environment

2.3.5: Xilinx Embedded Development Kit (EDK)

The Embedded Development Kit (EDK) provides users with a way of programming and using the onboard microprocessors that are on the development board [20]. The Vertex-II comes with two IBM PowerPC processors as well as the required cores for development of a MicroBlaze controller. The EDK allows for development on either platform. Use of the Xilinx EDK will be required for any microcontroller development related to self – reconfiguration. Although self reconfiguration may be completed through the development of custom logic, use of the PowerPC allows for an added level of scalability for project changes. The EDK is used

to synthesize all peripherals required for operation of the PowerPC as well as the development of embedded applications. This software operates as a separate entity but provides the possibility to export its designs in a format readable by Xilinx ISE as well as PlanAhead. The EDK significantly raises the complexity of the design flow and is expected to cause difficulties with the integration of different flows. A screen shot of the EDK environment is shown below in Figure 2.5.

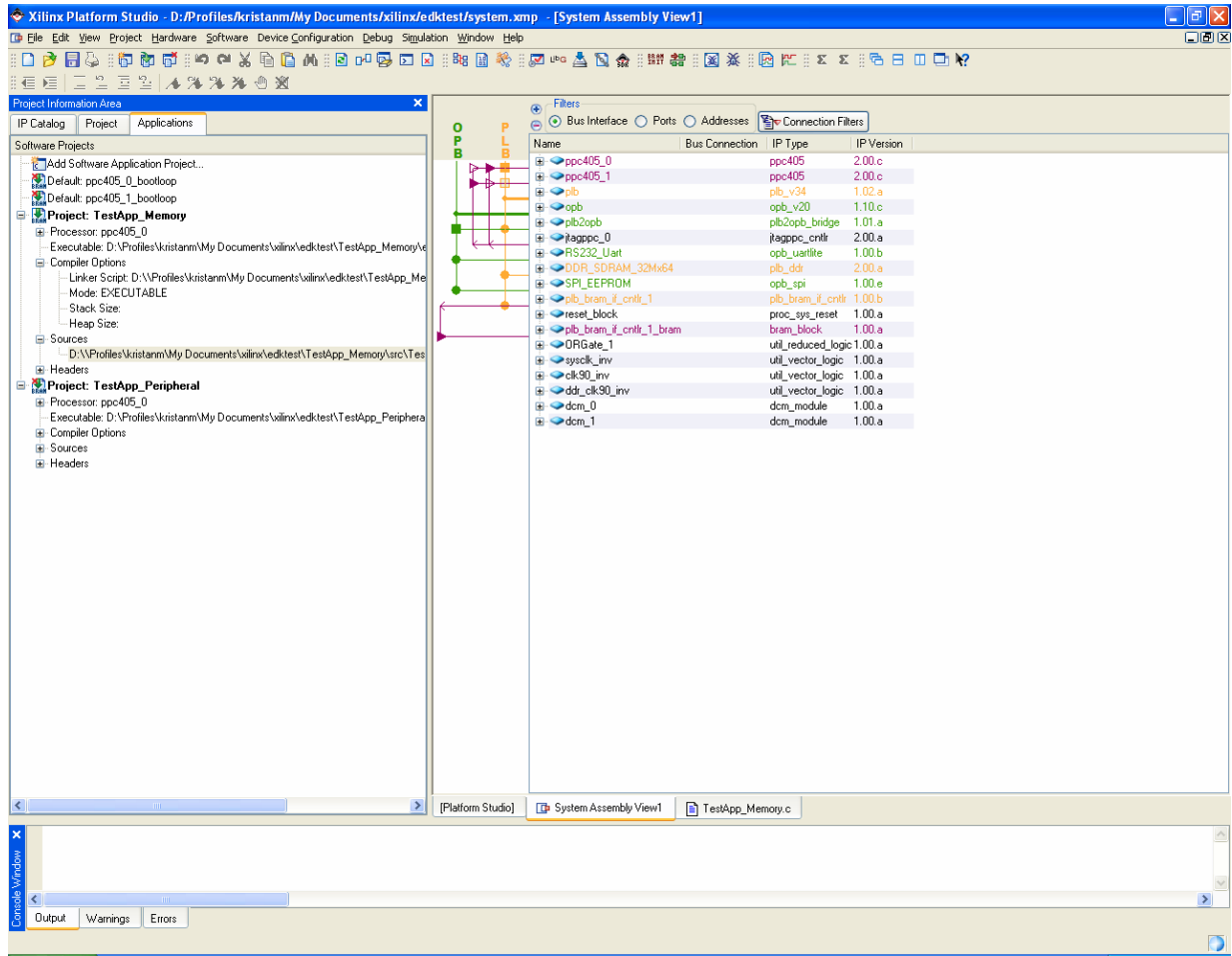


Figure 2.5 – Xilinx EDK Design Environment

2.3.6: Rational ClearCase

ClearCase is a source control management and revision system that is very popular in large development environments with multiple people contributing. With ClearCase, files are stored on a server in a repository. Users can then check files in and out of the repository and make changes. If a user does not want to work on the main development line and instead isolate him or herself from other check-ins a branch can be created. Once work is done in a particular branch, those changes can be merged back into the main line, or the branch

can be abandoned. Figure 2.6 shows how ClearCase graphically shows relationships between each branch and each checked-in element.

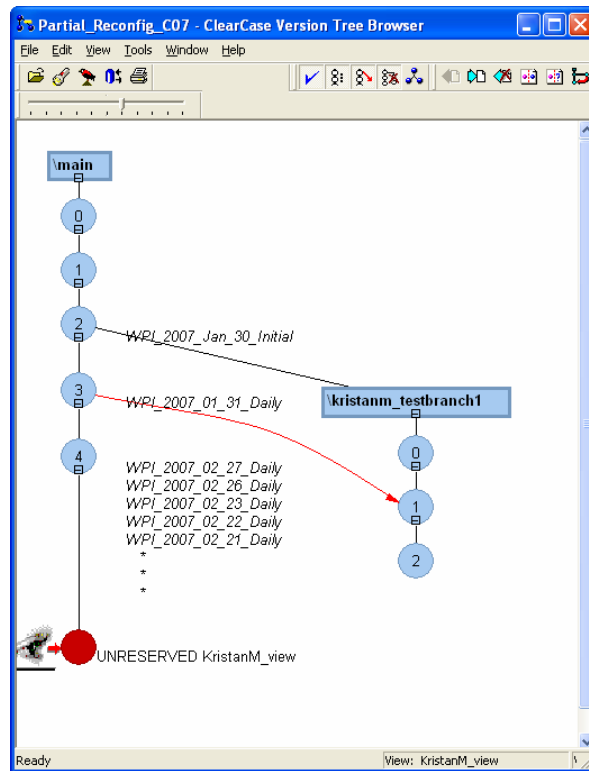


Figure 2.6 - Version tree of an element

ClearCase is powerful because in addition to controlling check-ins and check-outs, it will merge changes that happen between multiple users should multiple copies of a document be checked out at the same time. The differential viewer utility in figure 2.7, highlights changes between versions of a file within ClearCase.

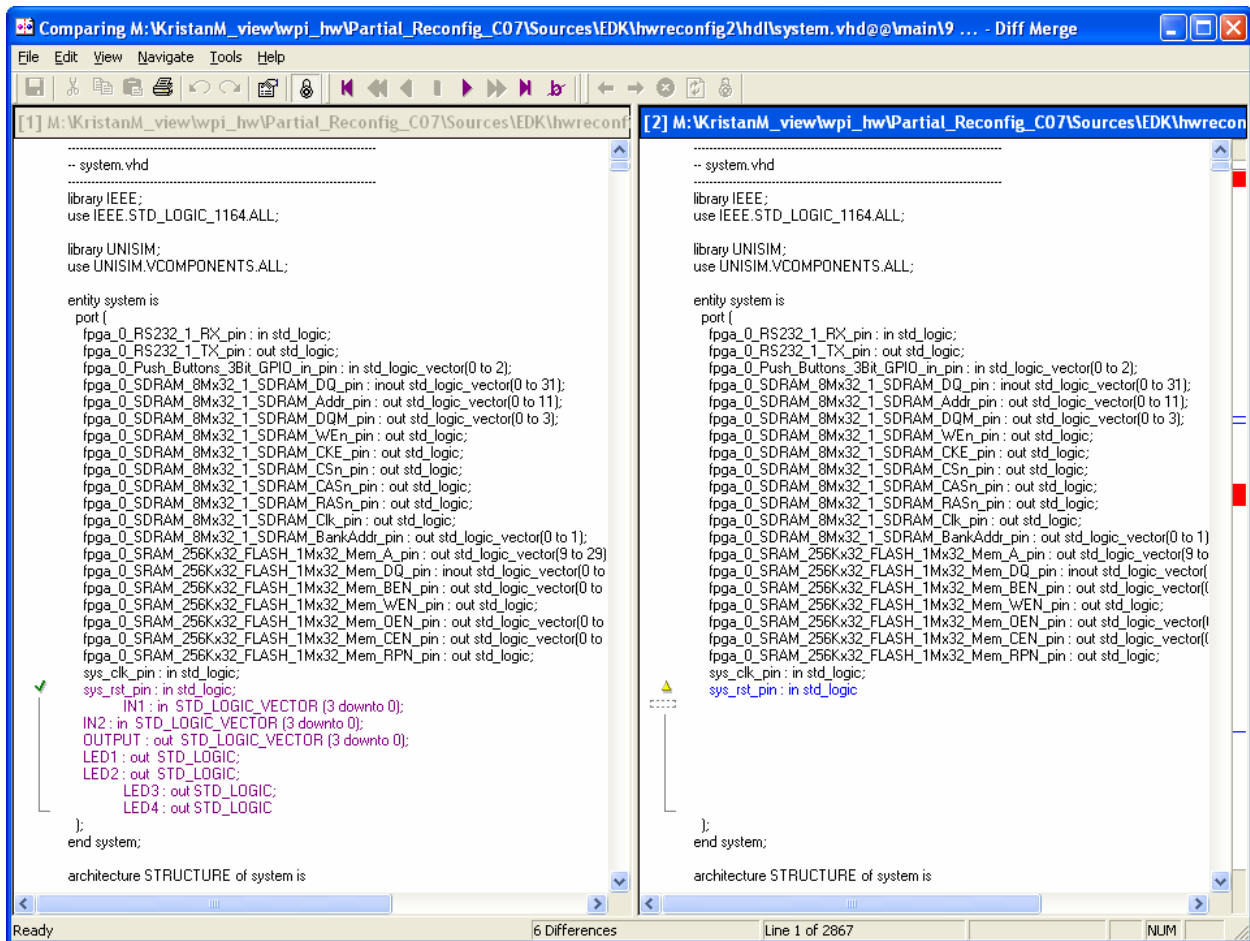


Figure 2.7 - Diff viewer showing changes between versions

2.4: Previous Work

This project draws from coursework and previous Major Qualify Projects completed with General Dynamics C4 Systems. Most notably, the project entitled “Efficient AES Implementation with Modes of Operation” completed in March of 2002 and “A Self Healing Circuit Implementing TMR” completed in April of 2006. Although this project does not directly deal with Triple Modular Redundancy, it is meant to lay a foundation for future work in that area. Partial Reconfiguration and TMR can be implemented together for increased reliability and stability of an logic system. Ultimately, the implementation of a modern encryption algorithm, such as AES, within a TMR Partial Reconfigurable system is the goal. This project however is limited in scope to the creation of a partially reconfigurable design.

2.5 : General Dynamics C4 Systems (GDC4S)

General Dynamics C4 Systems is a subsidiary of General Dynamics Corporation in Falls Church, Virginia. C4 Systems falls under General Dynamics Information Systems & Technology line of business. General Dynamics

C4 Systems is a leading integrator of network-centric solutions. Their leadership credentials come from applying world-class capabilities to create high-value, low risk solutions for use on land, at or under the sea, in the air and in space. Based in Scottsdale, Arizona, General Dynamics C4 Systems employs approximately 11,000 people worldwide and is focused on the development, design, manufacturing and integration of secure communication, information and technology solutions [7].

General Dynamics is the sponsor of this research and has graciously provided office space, software, and hardware in order to complete this project. Although General Dynamics is a government contractor and deals with classified information on a daily basis, all of this work was conducted in the public domain.

2.6: Advanced Encryption Standard (AES)

Advanced Encryption Standard (AES) was adopted by the United States government on November 21, 2001 and became effective as of May 26, 2002 [6]. AES effectively replaced the previous Data Encryption Standard (DES). Initially designed by Joan Daemen and Vincent Rijmen, the Rijndael algorithm was chosen by the National Institute of Standards and Technology (NIST) to be used for AES. AES has seen widespread use and is an extremely common form of encryption among software and hardware applications. Unlike its 56 bit predecessor DES, AES utilizes a 128 bit block structure with key sizes in 128, 192, and 256 bit forms [5]. In perspective, the 128 bit key size AES algorithm provides over 10^{21} more possible keys than a 56 bit key size DES algorithm [1]. This large key size causes AES to be extremely difficult to crack when using brute force methods. In its 4 years of existence as the government encryption standard, AES has been utilized all over the nation and world for both government and non-governmental use. AES owes its success due to its low cost, high speed, high security and low memory requirements.

Dating to 2006, the only successful attempt at breaking an AES system has involved side channel attacks [10]. A side channel attack is different from an attack based on a weakness in the algorithm itself and instead utilizes leaked information from the system itself. This type of attack is often based on timing information or transmission of leaked electromagnetic data and may require internal knowledge of the hardware system. Timing and power monitoring attacks involve physical monitoring of data flow and power usage through a system CPU and can be used to determine the size of the key being used for encryption. Data can also be leaked through radio waves generated from the encryption system. This vulnerability however can be easily safeguarded by the use of physical shielding for the device. In all cases of side channel attacks, physical access to the system is often required and can be defeated with better computer security procedures. Advances in hardware technology continually strengthen the security of these systems and allows AES to remain a secure cryptographic system.

2.7: Triple Modular Redundancy (TMR)

Triple Modular Redundancy or TMR is the concept of duplicating a black box circuit three times and using the output that appears in a majority of the black boxes. TMR was first proposed by Von Neumann in digital circuits that have binary output and is shown in Figure 2.9. TMR systems are setup by having a black box circuit or module (denoted M) replicated three times [21]. These outputs are then fed into a voting circuit. The voting circuit (denoted V) compares the three outputs and determines the winning result by taking the most popular result. The result from the circuit V is then passed along to the next device that relies on the output of M .

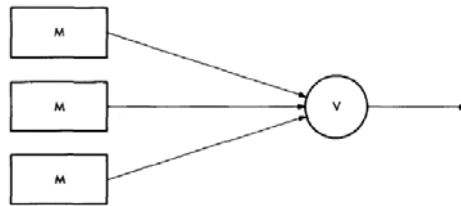


Figure 2.9 – Triple Modular Redundancy Description

There is one big assumption to this using a basic TMR system as described above. According to [21], the major assumption is that the voting circuit V is perfect and will not fail. In order to mitigate the risk of the failure of the voting circuit, the voting circuit must also be tripled and made redundant. Figure 2.10 shows a TMR system where the voting system is also redundant. The result of each voting circuit is then uniquely fed into the next component which is also replicated three times. The process continues throughout the entire system passing from one module to the next.

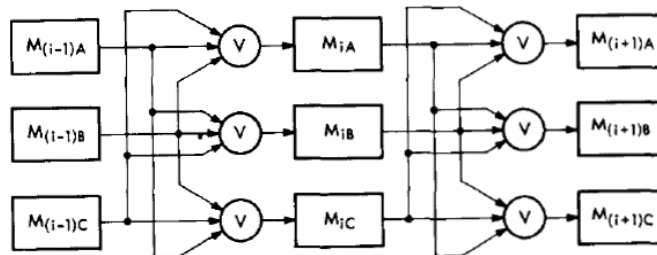


Figure 2.10 – Triple Modular Redundant System Description

By using a redundant system, the risk of an overall system failure is reduced because a TMR system can withstand an individual failure of a component. If a black box circuit has a reliability of R , a TMR system's reliability can be shown as:

As long as the reliability has a probability of .5 or greater, the reliability of the TMR system will be greater than the individual system as described [21]. An odd number of replications are chosen because this allows for a majority to always occur in a vote. Having an even number of components introduces the risk of having a tie [21].

TMR is a chosen solution for implementing a highly reliable circuit because it has been in widespread use long enough to prove that it is useful. Furthermore, the use of TMR has been successfully implemented in systems that are exposed to harsh environments and situations where circuits are very likely to fail and cannot be maintained easily. Examples of these circuits can be found in the PANSAT satellite systems designed by Worcester Polytechnic Institute. Given that systems in space need to work without any user intervention, a reliable system needs to be in place to ensure that data is processed correctly.

The other argument for TMR's use is that TMR is easy to implement. Once a module has been designed, it simply has to be replicated three times. Even though there is a small time investment associated with building the voting circuit, TMR designs are quick to build at least in theory.

2.8: TMR, AES, and PR Together

The basic foundation of this project as well as future projects involves the coordination of Partial Reconfiguration with Triple Modular Redundancy design methodologies. Ideally, these two technologies could be used to implement a stable, self healing AES implementation as well as any other useful encryption algorithm. Through use of an internal or external microprocessor, the system could be designed such that when the TMR voting circuit detects an error, the FPGA will partially scrub itself without interrupting the flow of data. This could eliminate encryption errors altogether as well as any downtime or need for human interference. The ideal result would be a completely self sustained system with extraordinarily high reliability.

Although TMR could be realized without the implementation of PR, system downtime would be required in order to scrub the circuit. In some cases, downtime may be an extremely undesirable circumstance. For example, if a non - PR, TMR design begins abnormal operation, the voting circuit would detect the bad output and discontinue use of that circuit. The entire design could then be realized to either scrub entirely at a certain time interval through use of the internal configuration port or set for scrubbing via user I/O. If a PR design were realized, the system would then only need to scrub the "broken" logic when necessary. This would eliminate the need for user I/O or a regular scrubbing interval. Although, both of these features could still be implemented if desired.

Partial Reconfiguration brings these new advantages to encryption implementation with the use of an FPGA. Increased reliability and the implementation of a self – sustaining system are further evidence that hardware encryption techniques are further evolving for any algorithm type.

3: Design Requirements

The basic design requirements for the initial working prototype are a direct result of thorough background research and identification of a basic project. The requirements serve as an abstract methodology that lays a foundation for proceeding forward. Initially, modular design flow had to be investigated, along with its relation to partial reconfiguration. Next, the design of the test circuit and the ways which self-partial configuration will be implemented had to be identified. Finally, the overall design flow for synthesis and implementation is investigated.

3.1: Modular Design Flow

Creation of a partially reconfigurable circuit within the Xilinx ISE development environment requires the designer to use a modular design flow [17]. Modular design is often used when a project requires multiple team members to implement a design. More importantly, it permits for development of several pieces of a design simultaneously, saving time, and avoiding problems associated with more complicated circuits [15]. Separate modules can be designed, repaired, and implemented separately. This allows the designer to change one module without affecting another and creating a chain of unwanted dependency problems. The required modular design flow is described below. Steps diagrammed in parallel can be completed simultaneously [17].

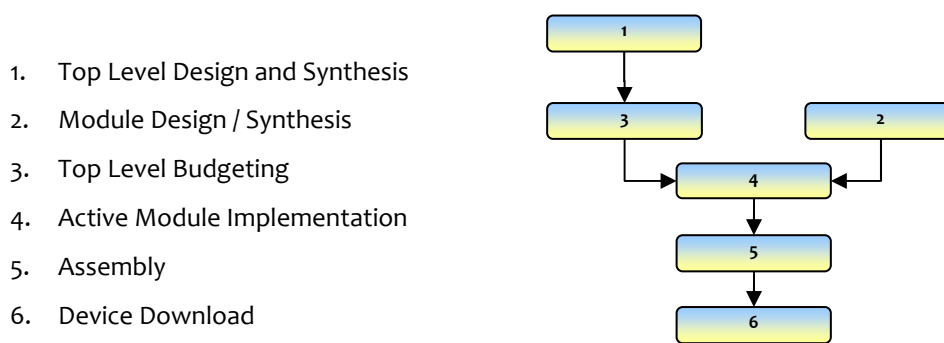


Figure 3.1 – Modular Design Flow

This design flow must be followed in order to successfully complete a modular design. Basic steps must also be followed during each design phase for a successful design. The lead designer must create the top level design as well as initial budgeting and assembly steps. A top level design provides the input and output requirements for each module as well as allows the designer to move ahead with timing, location, and physical area constraints for the final design [9]. This can greatly reduce the amount of time required for a start to finish design.

3.2: Modular Partial Reconfiguration Flow

Similarly to the modular design requirements previously described, partial reconfiguration also requires designers to utilize a specific modular design flow. This modular partial reconfiguration flow is very similar to the modular design flow described above with some added design requirements. These added requirements are detailed below [4].

- Area range constraints for reconfigurable modules must be applied
- Bus macros are required communication across reconfigurable modules
- Implementation must occur for each version of a reconfigurable module
- Assembly must occur for each combination of reconfigurable and static modules
- Full and partial bit files must be created for device download

When creating a partially reconfigurable design, it is necessary to assign specific area constraints to any reconfigurable modules. This is done in order to differentiate which area of the FPGA will be rewritten by a partial bit file [4]. This task is traditionally completed with the aid of Xilinx Floorplanner. However, the Xilinx PlanAhead software has integrated floor planning, macro placement, module implementation, and assembly into a single GUI. The difficulty in creating a partially reconfigurable design is greatly reduced by the integration of these tasks into a single tool. PlanAhead replaces the use of Floorplanner, Pace, Constraints Editor, and FPGA Editor, as well as provides a GUI for NGDBuild, Map, and Par commands [13]. The PlanAhead software allows the user to avoid using multiple programs while scripting the more complicated assembly and implementation stages.

3.3: Test Circuit Overview

3.3.1: Preliminary Requirements

Currently, partial reconfiguration is mostly limited to academic use and is only beginning to gain entrance in commercial applications. Xilinx ISE 8.2i does not officially support the partial reconfiguration design flow. However, there exists an early access toolset through Xilinx's protected website <http://www.xilinx.com/support/prealounge/protected/>. Because partial reconfiguration is a relatively unused practice, there exists little documentation on successful designs. In order to minimize errors and simplify troubleshooting, a preliminary test circuit should be designed for partial reconfiguration implementation. The test circuit must be able to limit complexity, provide I/O for testing, and be reconfigurable in nature. The development board provides eight DIP switches, eight output LEDs, and four push button switches which are ideal for basic testing. The design should include at least one reconfigurable and static module, both of which

should provide output to the user and be independent of each other. For example, a static module with constant output to an LED would allow the user to test for errors while a separate module is reconfigured. The reconfigurable module would only need to provide different operations depending on which version is downloaded to the board. Adder and subtractor modules are ideal for this as they can utilize the same I/O ports and complete recognizable operations for the user.

3.3.2: Programming Methods

Once the initial design is completed, there exist several methods of programming the full and partial bit files to the FPGA. The simplest involves programming over the JTAG chain like any standard (not using partial reconfiguration) design. Initially, a full design is loaded which includes any one version of the reconfigurable module. When a change is required, a new partial bit file is then downloaded which only changes the reconfigurable module while leaving the remaining modules operation uninterrupted. This method should be used for initial programming and testing as it reduces the complexity of the project.

Unfortunately, partial reconfiguration over JTAG is not feasible in a final product as it requires both a full computer and a specialized JTAG tool. Although almost all Xilinx FPGAs support partial reconfiguration, we need to develop a system where the FPGA can reload its own configuration, either from external memory or from data within the bit stream itself. Therefore, we will use the ICAP (internal configuration access port) port, which is available on Virtex series devices. Although it may be possible to perform a JTAG scrub using the platform flash, controlled reconfiguration is preferred. This allows for several configurations to be loaded, rather than simply scrubbing the circuit with the already existing configuration.

To accomplish this, it is necessary to develop an embedded system within the FPGA itself and use the ICAP to reconfigure it with pre-stored configuration files. There are two directions that may be taken, either using custom logic or through the use of a microprocessor system. The microprocessor has the distinct advantage of the fact that Xilinx provides the custom logic necessary to interface with the ICAP, so working with it is done at a high level using the C programming language. The disadvantage of the microprocessor system is the added complexity and system overhead that may not be necessary for this design. However, the microprocessor also allows the design to be far more scalable to larger reconfiguration designs than the custom logic design, and allows for a simple interface to many additional peripherals (such as SDRAM, Flash and RS-232) without requiring the time to write the VHDL for the custom logic implementations. In addition, the Virtex-II Pro device includes two hardware core 32-bit PowerPC microprocessors built in. Therefore, the processor itself does not require any FPGA resources; only the interface peripherals will use FPGA resources.

In addition to the choice between custom logic and microprocessor control for the ICAP, a decision must be made on where to store the partial reconfiguration bit files. The two primary choices are within the FPGA bit stream itself, or in external flash memory. The Memec Virtex II – Pro XC2VP30 – FF1152 development board includes a P160 Communications Module 2 add-on. This additional module includes 4 MB of flash memory and can be used to store the partial bit files. Use of the FPGA bit stream is much simpler, as it does not require programming the on-board flash memory or the expansion module. Additionally, it will work on any board using this FPGA. However, there is a distinct disadvantage to using the FPGA bit stream – in that it uses up the FPGA’s resources. This eliminates one of partial reconfigurations major design advantages of implementing large configurations on less capable FPGA’s. However, the 4 MB flash memory provided with the P160 module is able to hold at least 2 complete FPGA configurations (and many smaller partial configuration modules). The flash memory can also be used for storage of the code for the microprocessor. This further reduces the FPGA resource requirements because the only code required in the bit stream is a boot-loader to load the code out of flash memory.

3.4: Process Overview

3.4.1: Design and Synthesis

Modular partial reconfiguration requires the user to follow a specific design flow. This flow is similar to the more common modular flow but has added requirements for partial reconfiguration. The earliest step for PR design involves HDL design and synthesis. The designer may use his or her preferred design method and synthesis tool for this task.

Design and synthesis involves several added requirements. When coding at the top level it is necessary that all logic be constrained to “black boxes”. This means that the top level must only include I/O logic, “black boxed” modules, and bus macro instantiations. Creating a “black box” module in VHDL means that the designer neglects to include the source VHDL for the module during top level synthesis. This step is important for the following initial budgeting stage. Each module must be synthesized separately in its own project and each reconfigurable module must use the same number of input and output ports. Reconfigurable modules must pass all I/O data through bus macros, ensuring that no data paths are lost during reconfiguration. Before synthesis, it is necessary that the designer include Bus Macro sources, and remember to not include any other module source. Finally the designer must ensure that the synthesis tool does not add I/O buffers during module synthesis and that hierarchy is preserved at all levels. Completion of this step produces top level and module net lists which will be used in subsequent steps.

3.4.2: Budgeting and Constraints

The following steps can be created with a number of different tools. Xilinx ISE provides command line tools and scripts which can be used to complete the following steps. However, Xilinx PlanAhead integrates many of these tools into a single GUI interface. Once a new project is created in PlanAhead it is only necessary to import the top level net lists as well as all static and one reconfigurable net list to begin initial budgeting. PlanAhead must also be configured to utilize the PR functionality. This can be done by entering the console command:

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

Constraints may be added manually via a text editor or graphically in PlanAhead. There are several items which must be remembered when budgeting a partially reconfigurable design. These steps are outlined below and are required for successful PR implementation.

- 1) All static logic must be confined to a single base p_block with no range defined
- 2) PR regions must be confined to separate p_blocks and assigned a range
- 3) PR p_blocks must be rectangular and include the following constraints:
 - a) Mode = Reconfig
 - b) Min_x and Min_y coordinates must be positioned on an even slice
 - c) Max_x and Max_y coordinates must be positioned on an odd slice
 - d) BRAM ranges must also be assigned
- 4) Bus macros must be placed across PR region boundaries
- 5) I/O ports must be mapped to desired pins
- 6) DRC checks may be run to uncover planning errors
- 7) Floor-planned net list must be exported to a new (empty) directory.

Following these requirements will allow the designer to advance to the implementation stage of the design. Note that these steps only budget for an initial design. To budget subsequent reconfigurable designs, the user must only update the net list of the reconfigurable module they wish to change and export again to a new directory. The remaining design steps must then be carried out for each design.

3.4.3: Implementation and Assembly

The remaining design steps can be completed by running the “Run Partial Reconfig” tool in PlanAhead. The user is provided with the option of generating a batch file to complete all remaining steps or the user may run through each step graphically. When running the “build all” command it is not possible to provide separate parameters inside of PlanAhead, instead the designer must edit the batch file manually. This process is

generally faster and requires less input. However, more complex designs may require significant changes, in which case the graphical interface may be more suited for the task. As the implementation and assembly steps are run, PlanAhead creates a directory structure. Full and partial bit files for each design are provided and the designer can then download the files to their development board and proceed with testing. It is important to remember that these steps must be run for each version of a reconfigurable module to create all desired partial bit files.

3.4.4: Download and Testing

Successful completion of the previous steps results in the creation of full and partial bit files. Any full bit file can be downloaded to the Xilinx FPGA as the initial design. Similarly, any of the reconfigurable versions can be downloaded at any time. The number of bit files and file sizes is entirely dependant on the design. For an initial proof of concept, significant testing will be required to ensure no glitches occur. Functionality can then be observed by the user depending on the type of circuit created. As an initial proof of concept, it is important to implement a design which can be easily tested and repaired.

4: Implementation – Basic Circuit

4.1: Software Requirements

Creation of a partially reconfigurable circuit requires that the designer follow a set of strict guidelines for modular design. Proper synthesis of the design also requires that the software be properly configured for the partial reconfiguration design flow. Due to the intricacies of software variations, it is important to note that this design was created using Xilinx ISE 8.2i with Service Pack 1 and Early Access PR tools installed. Following synthesis, implementation of the design was completed using PlanAhead version 8.2.7 with Partial Reconfiguration tools enabled. While it is certainly possible to implement this design using other software versions, changes in design approach would be inevitable.

4.2: HDL Design and Synthesis

4.2.1: Design Overview

The partial reconfiguration design flow requires that the designer implement any static design separately than any reconfigurable portion. This can be done by separating logic into modules and using a top down approach for design. For this design, only two versions of the PR module and a single static module were created. It is possible however to implement several static modules and PR modules as well as multiple versions of each PR module. The top level design and each module must be created in a separate project and synthesized separately. Note that in this design 4 bus macros were used to ease routing from the reconfigurable module to I/O ports. This will be further explained later. This creates a separate net list for each module which will be used later during budgeting and implementation. VHDL code for the top level and subsequent modules are available in Appendix A. A basic overview of this design is shown below.

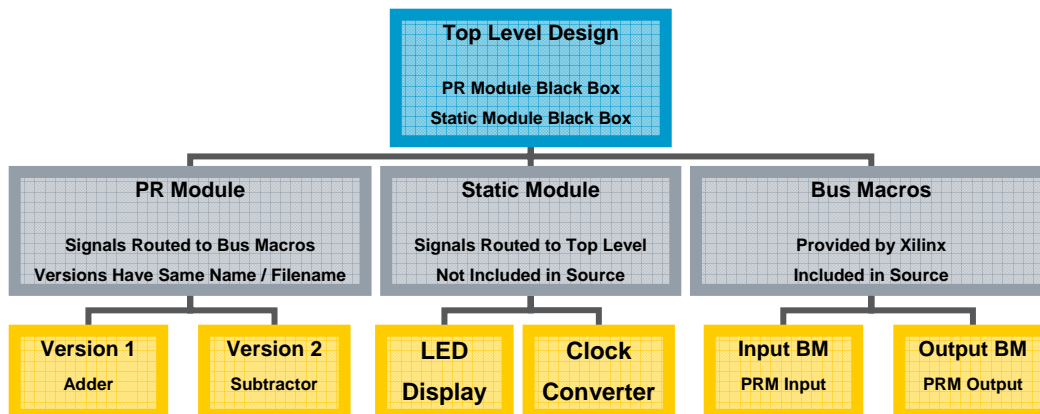


Figure 4.1 – Top Level Design Hierarchy

4.2.2 Design Intent

This partially reconfigurable design was developed as a proof of concept. The intent of this design was to create a partially reconfigurable circuit which is easily tested and would limit design problems. A design was chosen which would utilize only one reconfigurable and static module. A basic ALU would provide an easily testable design which could be written in a partially reconfigurable manner. In the interest of time the design was limited to only two operations, addition and subtraction. Each operation would be contained within its own reconfigurable module and a partial bit file would be loaded to the board to switch between the two. For testing purposes, a static module would control otherwise unused LEDs to test for errors during reconfiguration. Overall, this would require creation of four separate Xilinx ISE projects. The top level, static module, and PR modules would each require their own synthesis.

4.2.3: Top Level

As mentioned previously, HDL design at the top level must follow a strict set of guidelines for proper synthesis. The partial reconfiguration design flow requires that no logic be implemented at the top level. This means that all designs must be contained within “black box” modules. Black box modules are created simply by not including the source VHDL in the top level project. The entity declaration remains and when synthesized Xilinx creates an empty module with the correct number of inputs and outputs. The module itself must then be loaded into its own separate project and synthesized there along with any lower level logic. The basic circuit designed for initial testing includes a single static module and reconfigurable module. Top level modules must include bus macro sources for proper synthesis. Bus macros are included as a static routing method for reconfigurable modules. They ensure that when a reconfigurable module changes, I/O logic can be properly routed to and from the PR module. Any communication flowing in and out of a PR module, with the exception of clocking I/O must be routed through a bus macro. In the top level design, the bus macro can be instantiated and included like a normal entity. Signals are then required to connect bus macros to top level and module level I/O. As with any non reconfigurable design, all top level I/O must also be declared in the top level VHDL.

Once coding at the top level is complete, synthesis may begin. Any compatible synthesis tool may be used, however this design was synthesized using Xilinx XST. However, before synthesis can begin there are a few options which must be changed. Figure 4.2 displays the Xilinx synthesize process properties window. For synthesis at the top level you must be sure that **Keep Hierarchy** is set to Yes. To change this option, select the advanced view. Furthermore, under the Xilinx Specific Options tab check to make sure that **Add I/O Buffers** is enabled for the top level. These changes are necessary to implement and merge the final assembly.

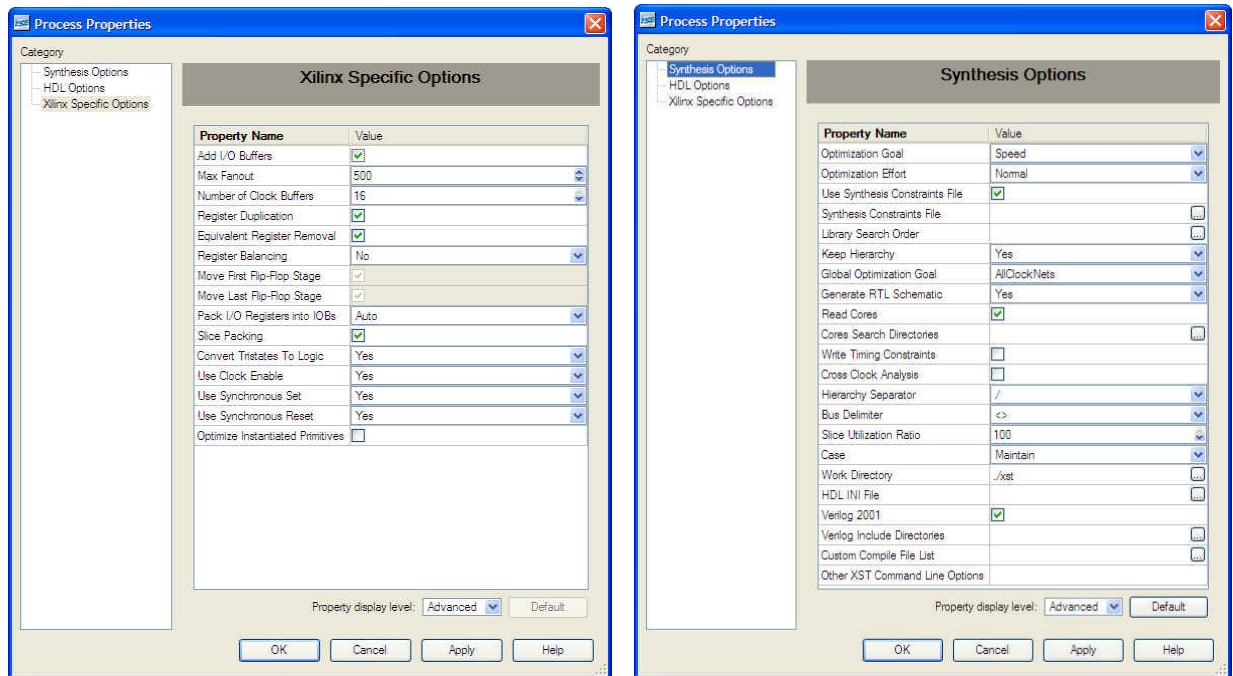


Figure 4.2 – XST Process Properties

4.2.4: Module Level

Module level design can be begun concurrently with top level design or top level initial budgeting. This process is for the most part independent of the others with the exception of a matching entity declaration required in the top level. There are a few things to remember when proceeding with module design for a PR circuit. Static module synthesis only requires that hierarchy be preserved as with the top level. Unlike the top level though, static and PR module synthesis require that I/O Buffers not be added during synthesis. Design of the PR modules also requires the designer to follow a basic design convention. All versions of a PR module must include the same number of inputs and outputs as well as use the same entity name and filename. Each PR module version should be saved in a different directory in order to differentiate between them. Module level design is generally basic and only requires the designer follow the previous guidelines.

This design utilizes reconfigurable modules which operate with addition and subtraction behavior. The code for these designs is very basic and does not include a carry. Each module includes two – four bit switch inputs and a single four bit LED output. No clocking logic is used inside the reconfigurable modules. The static module behaves as an LED driver to the remaining four board LEDs. A clock converter is included within the static module and is used to blink one LED at 2 Hz and another at 20 KHz. The remaining two LEDs are tied to logic high for continuous operation.

4.2.5: Bus Macros

Possibly the most important design requirement is the addition of bus macros for inter-module communication. As stated in the top level design section, bus macros are required for all I/O communication through a reconfigurable module. This includes information passed to other reconfigurable and static modules, as well as board I/O. Bus macros are provided by Xilinx and are FPGA model dependant. Four bit macros are available for download in the XAPP 290 application notes [23]. For this design, we chose to use eight bit bus macros provided in the Early Access PR Lounge [27]. Access to this lounge is currently restricted and you must contact Xilinx directly and apply for access. Bus macros are unidirectional and dependant on placement. This means that whether they act as an input or output to your reconfigurable module depends on the direction and which side of the module they are placed on. For the Virtex II Pro, bus macros are available in only right-to-left and left-to-right configurations. For example, a right-to-left macro placed on the right side of a reconfigurable module acts as an input, while a left-to-right macro would act as an output. In addition to bus macro direction, Xilinx provides asynchronous and synchronous versions as well as wide and narrow types. For this project, simplicity was desired and an asynchronous narrow bus macro was chosen. A total of four macros were used for this design, providing an input and output macro on each side of the reconfigurable module.

4.3: Implementation with PlanAhead

4.3.1: Project Creation

Upon completion of the HDL Design and Synthesis, Xilinx ISE creates several net list (.ngc) files. Several options for the designer remain to continue with the implementation phase of the project. Xilinx ISE provides all of the tools necessary for implementation of a PR design. However, these tools are divided into several GUI programs and command line scripts. Alternatively, Xilinx PlanAhead software can be used as a single integrated GUI for these tools. For this design, PlanAhead is used to complete the project implementation. Before importing the net lists into PlanAhead, it is beneficial to organize your directory structure of previously generated files. There may be a need to revert back to Xilinx ISE to make changes to the project and may not want to allow other programs to make changes to those files. The first step is to create a new directory for the PlanAhead project. When a new project is created, the software prompts the user to point to a location of synthesized net lists. It is acceptable to point to the original files since PlanAhead creates copies in its own directory folder when files are imported. For the first design, it is necessary to locate the top level net list, static module net lists, and only a single reconfigurable module net list. PlanAhead will then prompt to import a user constraints file (.ucf). This step can be avoided if you wish to create all constraints through PlanAhead. The remaining step for project creation is to enable Partial Reconfiguration in PlanAhead. This can be done using the command provided earlier in section 3.4.2.

4.3.2: Budgeting and Constraints

Initial budgeting, floor planning, and creation of constraints can be completed within PlanAhead. The first step is to assign all logic modules to physical blocks. To do this, begin with all static modules. Highlight and select **New pblock**. All static logic modules can be assigned to a single pblock. In this case it is referred to as pblock_base. No further floor planning is required for the static logic modules. Following the assignment of static modules, the reconfigurable modules are added to their own pblock as well. This step is completed instead by selecting the **Draw pblock** command. Then, a point is selected on the package view and dragged out forming a rectangle which will be designated as the reconfigurable module region. This rectangle can be as large the whole design or as small as a single slice. It must however be large enough to fit the reconfigurable design within it, and small enough to allow space for static logic to be routed as well. Placement of the module can not be completed automatically by the PlanAhead software. For simple designs, a number of floor plans may be suitable. An example of an acceptable pblock size is shown in Figure 4.3. Note that this pblock is much larger than required for such a simple module design.

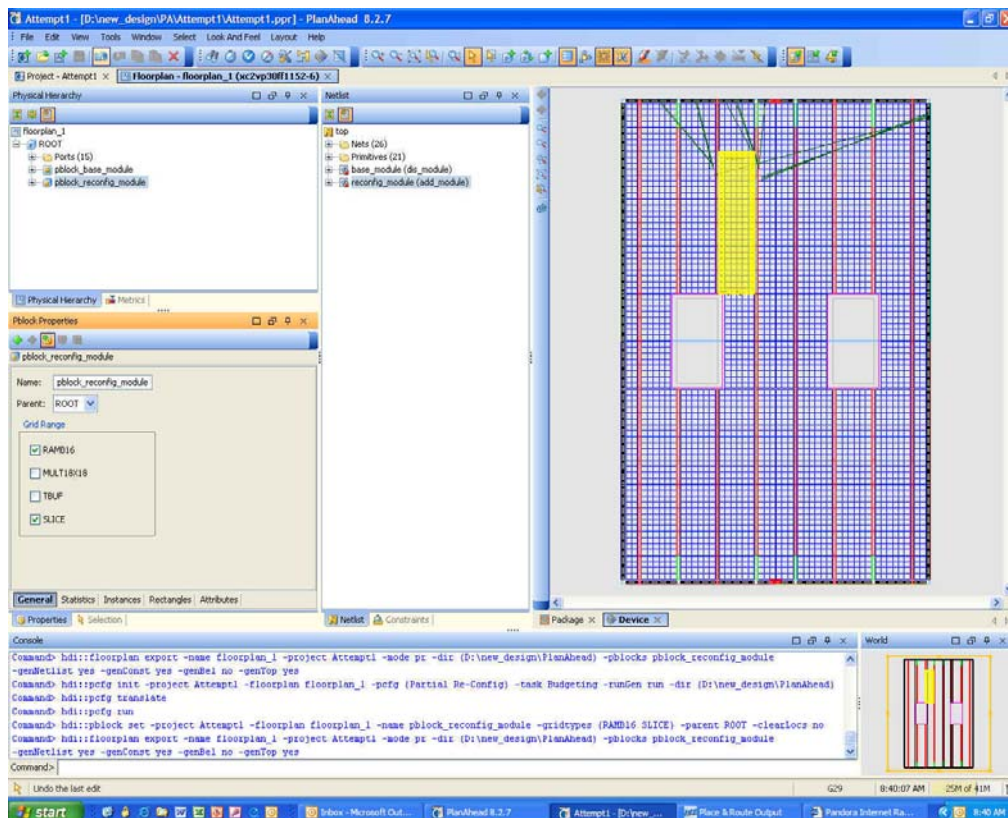


Figure 4.3 – Reconfigurable Module (Yellow) in PlanAhead Device View

PlanAhead has a built in requirement that only allows pblocks to be drawn in rectangular form. However, the user will also have to determine correct placement of the rectangle. Currently, PlanAhead requires that the minimum x and y values (upper left corner) of the module be placed on even numbered slices, while the

maximum x and y values (lower right corner) be placed on an odd numbered slice. This can be easily set by selecting the reconfigurable pblock, selecting rectangles in the properties window and changing the listed locations manually. Finally, select the attributes tab and add **DEFINE MODE = RECONFIG** to the list. This is required for proper creation of the bit files later on.

Once the pblocks are correctly defined, the bus macros need to be placed on the floor plan. This particular design utilizes four different bus macros. Each bus macro was designated in synthesis by its direction and input / output type. **inputbusleft** and **outputbusleft** need to be placed on the left side of the reconfigurable module, while **inputbusright** and **outputbusright** must be placed on the right side. When creating a design with bus macros it is important to note which side the bus macro should be placed because they are unidirectional and the type denotes whether they will function as an input or output. Naming the bus macro according to their location and type can be very helpful when floor planning. To place the bus macros, switch to **Site Constraint Mode** and collapse the top level primitives folder. Alternatively one can run a search for type PR bus macros. Select the macro and drag it into the device view. It must be placed so it straddles the PR region boundary. This occurs when the mouse is hovered over the lower left corner of a slice directly to the left of the boundary. Proper bus macro placement is displayed in Figure 4.4 with each bus macro highlighted differently.

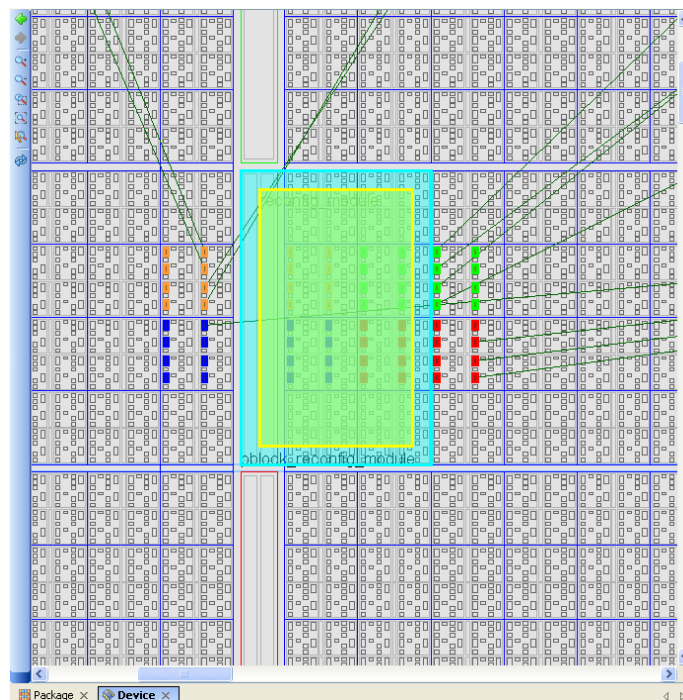


Figure 4.4 – Bus Macro Placement (Various Colors)

Once placement is completed, the last remaining step involves assignment of ports to I/O pins on the FPGA. Port assignment is entirely dependant on the development board and requirements of the design and can be

completed manually by adding the locations to a user constraints (.ucf) file or within PlanAhead. The constraints added for this design are provided below.

```
NET "CLOCK" LOC = AH17;
NET "IN1[1]" LOC = H22;
NET "IN1[3]" LOC = H21;
NET "IN2[1]" LOC = G19;
NET "IN2[3]" LOC = G18;
NET "LED2" LOC = E32;
NET "LED4" LOC = F30;
NET "OUTPUT[1]" LOC = E2;
NET "OUTPUT[3]" LOC = E4;
NET "IN1[0]" LOC = G22;
NET "IN1[2]" LOC = G21;
NET "IN2[0]" LOC = G20;
NET "IN2[2]" LOC = H19;
NET "LED1" LOC = E31;
NET "LED3" LOC = F31;
NET "OUTPUT[0]" LOC = E1;
NET "OUTPUT[2]" LOC = E3;
```

To enter these constraints in PlanAhead, simply open the package view, and drag and drop the ports from the Physical Hierarchy menu into the desired pins. PlanAhead will create the constraints file and add these values automatically. The final remaining step in the budgeting phase involves running design rule checks. Design rule checks are provided by PlanAhead to counteract any basic floor planning mistakes. Because the process involves so many steps, design rule checks can be very helpful to eliminate common errors. For this design, a common error was found when running design rule checks. This was due to the fact that we utilized asynchronous bus macros. Xilinx suggests that synchronous bus macros are used to eliminate any timing issues related to I/O from reconfigurable modules. For this basic circuit, timing issues are not of a concern and the errors produced by the DRC tool can be ignored. Note that if any other errors occur, it may be necessary to resolve them before moving on to the Partial Re-config Tool steps.

4.3.3: Partial Re-config Tool

The most important and useful part of the PlanAhead software is its ability to eliminate complicated scripting and command line use. The Partial Re-config Tool provides the user with a GUI of the remaining implementation and assembly stages. However, before this tool is run it is necessary to complete one more step. Select **File > Export Floorplan**, select a new, empty directory and select **OK**. This step creates the required directory structure for the Partial Reconfig Tool. Make sure that you export to a new or empty folder or the directory structure created will be incorrect. Once this is completed, the Partial Re-config Tool can be started. The tool provides two options, Run place & Route, or Generate Script Files. The first option allows each step to be run separately and provides a GUI for additional parameters to be added to each stage. The second stage creates a single batch file which once run completes all steps without any required interaction from the user. This step is fastest and recommended if no changes are required to the design. If this step is chosen, make sure that when the tool is initially run, the partially reconfigurable block is highlighted. This

causes that block to default as the reconfigurable block within the tool. If this is not done, the tool will not function correctly. If the first option is chosen, it is important to note that a bug within the PlanAhead 8.2.7 environment requires that the final assembly step be ran as generate script file. This bug was fixed in version 8.2.10.

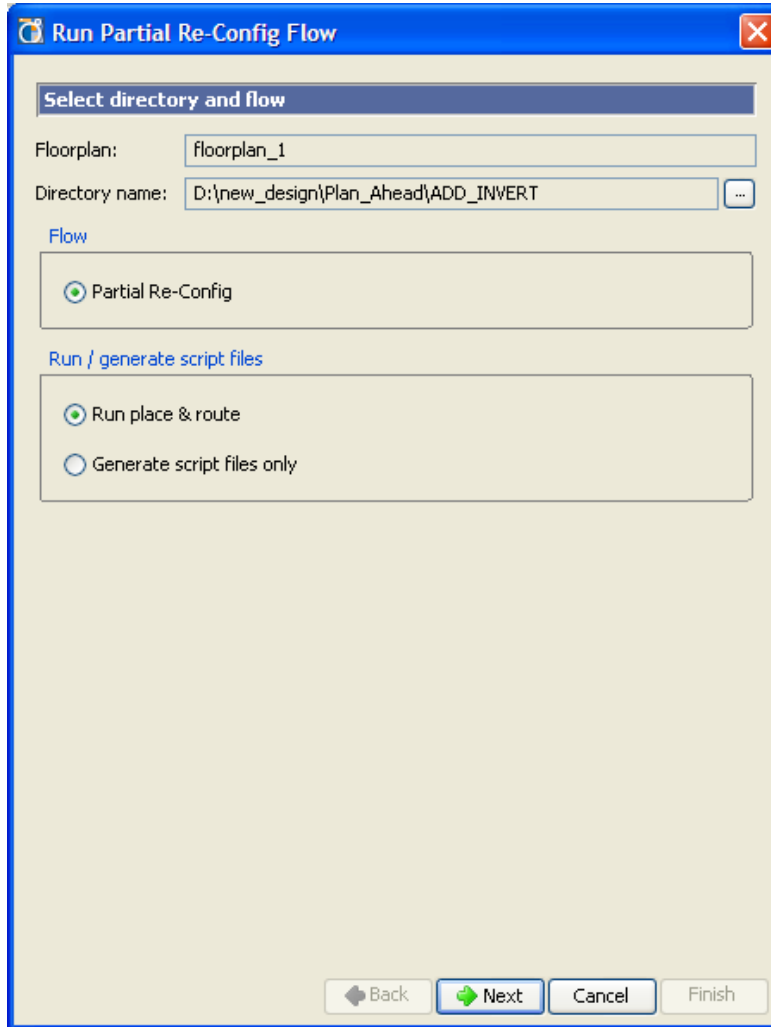


Figure 4.5 – Partial Reconfig GUI

An important requirement of the Partial Re-config tool is that each step is dependant on the steps previous to it. For instance, if the static implementation step is ran again, the following PR implementation and Assembly steps must also be completed a second time. Once the assembly stage is completed successfully, the bit files are generated and stored in the merges directory of the exported floor plan. These bit files are labeled according to the PR module implemented and can be downloaded to the development board. To initiate other PR versions, another PlanAhead project must be created. Then, import the net lists as before with a different reconfigurable module. The remaining budgeting and constraints steps can be completed by importing the user constraints file generated from the previous project. The Partial Reconfig tool can be run again following

the same steps as before. Be sure to choose a separate directory for floor plan export once again. Another set of bit files is created and the desired bit files can also be loaded onto the board. In order to distinguish the difference in bit files it is necessary to rename them in a manner that describes their behavior. Once implementation and assembly of each PR module is completed, programming the FPGA can begin.

4.4: Programming

4.4.1: Configuration over JTAG

To begin testing the design, the initial static_full.bit file was downloaded to the development board through the JTAG chain. JTAG is the simplest programming procedure completed through the Xilinx iMPACT software. However, JTAG programming requires a separate computer or board to load the files through the JTAG chain. JTAG is ideal in testing environments as it has no requirement of programming logic or added complexity. Operation “in – field” does not always allow for the use of an external programming device. Several options for this type of programming exist and will be explained later. To begin downloading, Xilinx Impact is run and the JTAG chain is initialized. This is done assuming the development board is properly connected to the PC via a USB or parallel Xilinx platform cable. Impact then examines the JTAG chain and locates the on board FPGA. Following JTAG initialization, the desired full bit file was selected and programmed to the FPGA. Note that any of the full bit files created may be downloaded initially, it is up to the designer which version is considered to be the initial setup. Partial bit files are then downloaded to the board in the same manner. When a partial bit file is selected to load, it is not necessary to power down the board or to re-initialize the JTAG chain. The FPGA will continue to operate non reconfigurable regions while the new partial bit file is loaded.

4.5: Creation of a PR AES Implementation

4.5.1: Design Overview

Similarly to the basic design described in section 4.2, a more complicated AES implementation can also be implemented through use of the ISE PR and PlanAhead tool flows. The previously mentioned circuit includes very limited I/O as well as a very small amount of reconfigurable logic. This meant that implementation of PR would involve minimum complexity. The AES implementation however is much more complex and will require significant changes in the design steps followed. The basic design is very small, so small in fact that the dominant size requirement in floor planning was the minimum number of slices to accommodate bus macro placement and to pass DRC checks. This meant the reconfigurable module would be a minimum of two slices wide and 4 slices high. The AES implementation however utilizes over 30% of available slices on the Virtex II Pro and would require a much larger reconfigurable module. To begin implementation, the design needed to be

modified for PR compatibility. The top level design would include two AES implementations and a switching circuit between the two. To keep complexity of the design at a minimum, this switching circuit would be designed to accept user input from an available push button switch. The user would be able to select one instance for use, while choosing to reconfigure the other via the JTAG port.

4.5.2: Limitations

It should be noted that the AES implementation involves an extremely large amount of I/O. It includes a 256 bit key input, a 128 bit data input and output, a 1 bit key load input, a 2 bit key size input, a 1 bit data load input, and a 1 bit data done output. Currently, AES implementation on the FF1152 dev board is not feasible without the use of a high speed serial or other interface which would be used to load and accept these numerous inputs and outputs. Instead, shifting registers can be used to limit the I/O to a much smaller number which can then be connected to available pins. This would still not allow for full testing but instead would allow for proof of concept with the basic PR implementation. The implementation can still be exported and floor planned in PlanAhead as well as developed into partial bit files.

4.5.3: Steps Required

The steps required for implementation of a PR AES design are very similar to those followed in section 4.2. First, a top level design must be created which includes only black box instantiations and bus macros for communication across reconfigurable modules. Due to size limitations, the AES implementation will only be able to be replicated twice on the chip. The shift registers used for I/O limiting and the switching circuit need to be created in a modular fashion and synthesized separately. Similarly to the early design, all modules must be synthesized in their own directory without I/O buffers and with hierarchy. The synthesized net lists can then be imported into PlanAhead and floor planning can begin. Note that the pblocks created for each reconfigurable module must be large enough to accommodate the code. This can be checked by selecting the general properties box and comparing the required and available statistics for slices and other attributes. Because a single AES implementation requires around 30% of the chip, each reconfigurable module will be quite large. Currently, PR design is also limited to placement only in different vertical columns. Although reconfigurable modules do not need to be the entire height of the FPGA, they cannot overlap or share vertical columns. With the exception of additional size and placement requirements, the steps involved in implementing the AES design in PlanAhead are the same as detailed for the basic PR design.

4.5.4: Feasibility

With the large size of this particular AES implementation, it is apparent that a TMR system with PR will not be feasible on this particular FPGA. The process however could certainly be realized with either a smaller AES implementation or larger FPGA. The development of the basic proof of concept leads us with reason to believe

that a PR AES implementation is entirely possible and could be utilized with a TMR system. Due to the apparent size restrictions and testing limitations, further development of this PR AES implementation was abandoned in order to focus on the self reconfiguration design described in section 5.

4.6: Testing and Results – Basic Circuit

4.6.1: Overview

Completion of the basic PR design requires that testing be completed to ensure proper configuration. This basic testing is required for two phases. First, the design can be tested through configuration over JTAG. This is the simpler of the two approaches and allows for the initial testing of the PR implementation. Once initial testing is completed, the self-reconfiguring design can be loaded and tested. In the initial design, several features were added in order to allow for testing. The LED display module was created for the sole purpose of visible testing the functionality of partial reconfiguration. This module uses four on board LEDs. The output to these LEDs can be monitored by the eye as well as more closely through the use of an oscilloscope.

The goal of a partially reconfigurable design is to change a portion of the hardware without affecting the performance of remaining architecture. With this intent in mind, the LED display module was created as a static, non reconfigurable module. The reconfigurable module implemented functioned as an addition or subtraction module. This module utilized an eight input DIP switch as well as four output LEDs separate from those used in the LED display module. If partial reconfiguration is successful, the reconfigurable module can be scrubbed, erased, and rewritten all without interruption to the LED operation.

4.6.2: Bit Stream Differences

As described in the implementation section, Xilinx ISE and PlanAhead were used to generate a number of bit files for this PR design. For initial testing, five bit files are of main concern. These files have been renamed to describe their functionality. *Adder_full.bit*, *Adder_blank.bit*, *Adder_partial.bit*, *Subtractor_full.bit*, and *Subtractor_partial.bit* are all required for initial testing. The full bit files include all static logic as well as the reconfigurable logic for which they are named. *Adder_full* is the full design with the addition logic as the initial reconfigurable module while *Subtractor_full* includes the subtraction logic. *Adder_blank* is the partial bit file with no logic included. When this file is loaded, no addition or subtraction logic should take place as the reconfigurable module is essentially erased. *Adder_partial* and *Subtractor_partial* are the two partial bit files for this design. They can be used to revert from one type of reconfigurable module to the other. Proper design will allow for these files to be loaded without disruption to the remaining static logic.

4.6.3: Desired Functionality – Configuration over JTAG

A successful implementation will result in specific expected outputs from the development board. Initially, as either of the full bit files is loaded, LED1 will operate at 2Hz, LED2 at 20 KHz, and LED3-4 at a constant on configuration. LED5-8 will be dedicated to the logic operation determined by the loaded bit file. The adder and subtractor both operate assuming DIP1-4 and DIP5-8 as two 4 bit inputs. The resulting arithmetic operation is output to LED5-8 with the carry bit discarded. All eight LEDs will provide the necessary output for testing both visible and with the use of an oscilloscope. As the partially reconfigurable module is scrubbed, erased, or modified, the four LEDs corresponding to the static display module should see no change in operation. An oscilloscope can be used to trigger on the falling edge of the output to ensure no lapse in operation takes place. The functionality of the reconfigurable module itself can be tested simply by moving the DIP switches and observing the outputs to LED5-8. Correct operation of the module can be confirmed if the two inputs are successfully added or subtracted from one another. Furthermore, when the blank bit file is loaded, no logic operation should take place and the corresponding LED5-8 should show no output values.

4.6.4: Results – Configuration over JTAG

Initial tests began with the loading of the *Adder_full* bit file. Operation of the hardware at this time was observed to be successful as the logic operations were correctly output to the proper LEDs and the static LED display module appeared to be operating correctly. The oscilloscope was used to verify the rate at which the LEDs were operating. To ensure that the results were conclusive, the *Adder_Full* bit file was loaded and reloaded several more times, each with the same operation results. An oscilloscope reading of the 2Hz and 20KHz signals is shown in Figures 4.7 and 4.8. Similarly, a reading of the solid LED signal in both on and off configurations is shown in Figures 4.9 and 4.10. Note that the LEDs operate on inverted logic. This means that when they LED is in an on state, the logic reading is low and in an off state the logic reading is high.

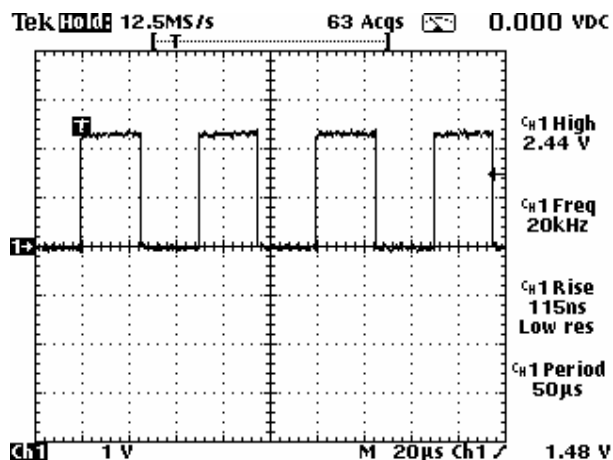


Figure 4.6 – 2Hz LED

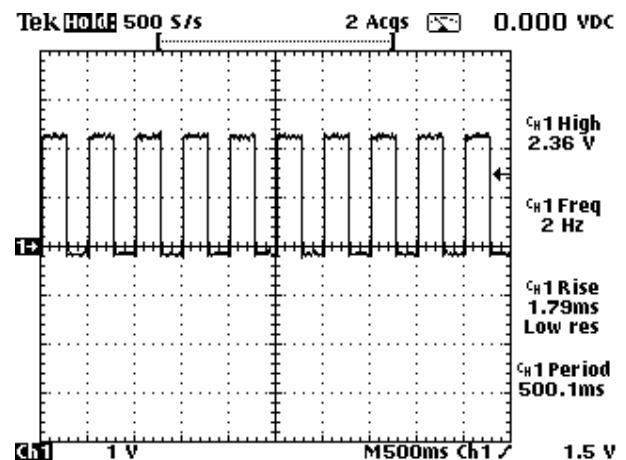


Figure 4.7 – 20KHz LED

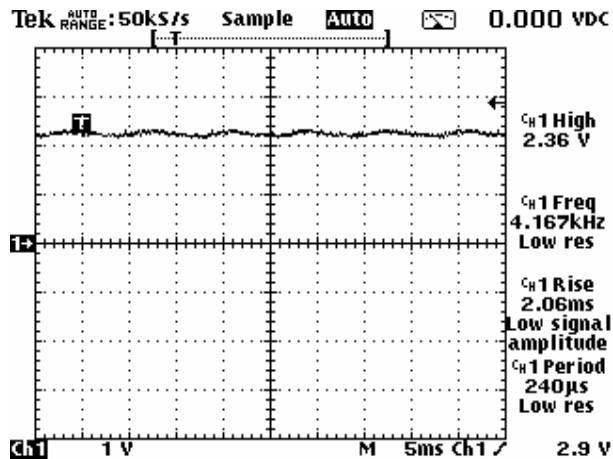


Figure 4.8 – Solid LED in Off State

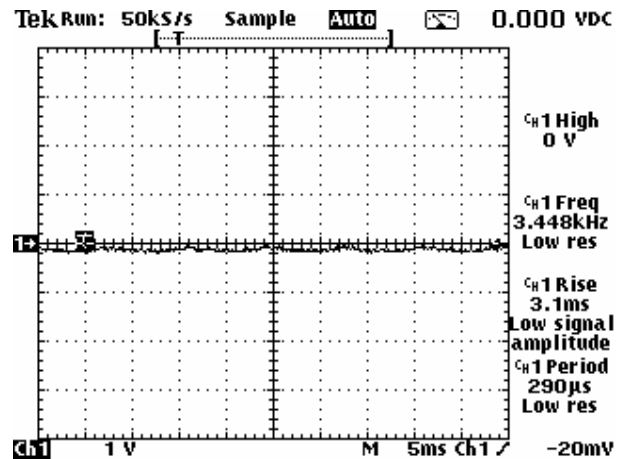


Figure 4.9 – Solid LED in ON State

The next step involved scrubbing the reconfigurable module by loading the *Adder_Partial* bit file onto the FPGA. This would essentially reload the adder logic already loaded onto the board. When this operation took place, the static LEDs were unaffected. The oscilloscope was set to trigger on the falling edge of one of the static LEDs. Essentially, if any loss of signal occurred the oscilloscope would catch it and pause the display for viewing. As the module was scrubbed, no visible effects occurred to the static displays. The oscilloscope was then used to monitor the output of the LEDs tied to the reconfigurable module. These in fact showed a slight change due to the quick scrubbing of the circuit.

The next step involved “erasing” the reconfigurable module by loading the *Adder_blank* bit file. When this occurred the same steps were taken to monitor the static display LEDs with the same result. However, the behavior of the LEDs tied to the reconfigurable module had now changed significantly as expected. Because the module was “erased” the output LEDs tied to it were all turned off. This demonstrated that the module was reconfigured properly. In order to test this with the oscilloscope, channel 2 was set to monitor one of the LEDs which would change state when reconfiguration occurred. Because the state of the LED would be changing from ON to OFF, the trigger was set to the rising edge. This would cause the oscilloscope to hold its value when the change occurred allowing us to view channel 1 at that exact point. Channel 1 would be connected to what should be an unaffected LED in order to ensure no change occurs. Figure 4.11 shows the 20KHz LED signal during reconfiguration. Note the point at which the edge rises on channel 2. There is no unexpected signal change at this point in Channel 1. Figure 4.12 shows the same approach taken, this time with a non oscillating LED signal on Channel 1. Once again, there is no unexpected signal change during reconfiguration.

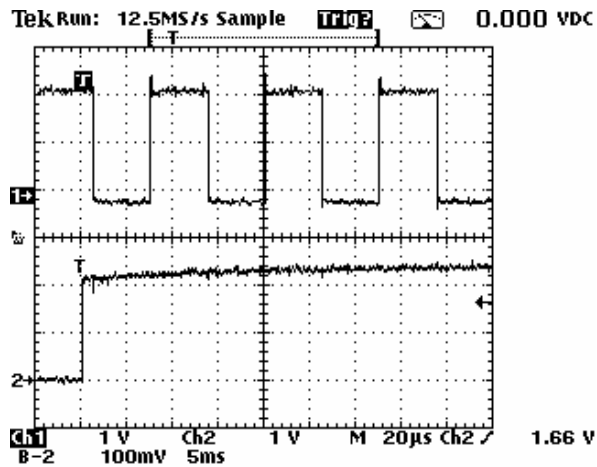


Figure 4.10 – 20Khz LED w/ Reconfiguration

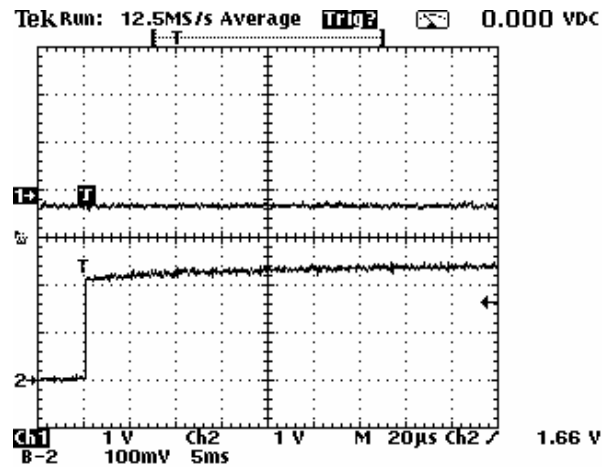


Figure 4.11 – Solid ON LED w/ Reconfiguration

The final reconfiguration step required was to load the *Subtractor_partial* bit file onto the FPGA. This would rewrite the reconfigurable module to utilize the designed subtraction logic. Once again, the oscilloscope was used to monitor the behavior of the static LEDs with the same result. This time, the reconfigurable module was once again configured properly as the LEDs tied to the reconfigurable module changed behavior. They now represented the output of a subtraction operation from the two 4 bit inputs. To conclude testing, these steps were taken again with the *Subtractor_full* bit file loaded. Results continued to be conclusive as no glitches were visible during reconfiguration.

5: Self-Partial Reconfiguration Implementation

Although dynamic partial reconfiguration opens the door for several new possibilities in FPGA design, external control of the reconfiguration process limits the usefulness of partially reconfigurable designs. The proof of concept for partial reconfiguration over JTAG demonstrated that dynamic partial reconfiguration is possible. There is still quite a bit to be desired in terms of usefulness in a final application. A design which could control self reconfiguration would eliminate the need for extra hardware reducing size and cost.

5.1: Overview

5.1.1: Advantages over JTAG Partial Reconfiguration

The primary advantage of using self-partial reconfiguration is that it allows for a complete “system on a chip” design, where no external control is necessary for the partial reconfiguration to occur. Some additional components are necessary, such as flash memory to hold the partial bitstreams and the FPGA’s original configuration, but all of the control circuitry for the partial reconfiguration logic is internal to the FPGA. Particularly, the method we used for basic JTAG partial reconfiguration required a full PC with iMPACT software and a Xilinx platform cable. This is not practical in a final application; although it is likely possible that a microcontroller could be configured to replace these devices.

A critical advantage to self-reconfiguration is the fact that it can be automated, rather than requiring a user to press the program button on the PC as is currently required. Although this could happen with JTAG and a microprocessor, it requires significant IO monitoring to occur, or a scrubbing circuit on a timer. Other advantages include easier portability to a triple mode redundancy design, as an internal reconfiguration controller can easily have direct access to signals used anywhere on the FPGA by simply changing the VHDL code. This means that the monitored signals can be changed just by updating the firmware, rather than requiring changes to a circuit board. In addition, this means that fewer FPGA resources are required for a TMR implementation, as all of the monitoring happens internally. Another related advantage is that it eliminates possible failures in the TMR circuitry, as the reconfiguration logic can reconfigure any FPGA routing, which is not possible with external routing on a PCB.

5.1.2: PowerPC vs. Custom Logic

As discussed in Chapter 3, the Xilinx Virtex series FPGA’s provide an internal port for reconfiguration by internal logic called the Internal Configuration Access Port (ICAP). Implementation of self partial reconfiguration requires that the designer choose to use custom logic or the internal PowerPC for controlling reconfiguration.

Use of the PowerPC holds two key advantages. First, Xilinx already provides the interface to the ICAP controller for reprogramming the FPGA, as well as some example C code for the PowerPC. In addition, interfaces to several other components are provided, such as RS232 for debugging and Flash memory for storage of the partial bitstreams. However, in a custom logic design, the logic necessary to control the ICAP port and the interface to the flash memory would have to be designed from scratch. The second key advantage to the PowerPC is its scalability for larger designs. Because it is programmed in C code, and is a programmable microprocessor rather than custom logic, it can scale much more easily to adapt to different applications. Due to the fact that this project is not research for one specific design, adaptation of the PowerPC was the more useful approach.

One major disadvantage of the PowerPC, is the fact that the peripherals (such as the flash memory controller, the RS232 controller, and the ICAP controller) are not as resource efficient as possible, as they are designed for general purpose use and may cover cases that we will never encounter. A custom logic interface would only have logic necessary for our specific application. It should be noted that the PowerPC itself, on our Virtex 2 Pro boards, does not require any additional logic as it is a hardware device that does not use reconfigurable resources. Finally, if it was desired to have TMR principles applied to the self-reconfiguration controller, there are only two PowerPC processors on the specific Virtex 2 Pro model that we are using, which means that triple redundancy of the control circuits is not possible. However, there are versions of the chip with four PowerPC modules, and it is unlikely that the PowerPC itself needs to be monitored for faults as it does not include any reconfigurable logic.

5.2: Design Procedure

5.2.1: Overview

We separated the design of the self-reconfiguration controller into distinct steps. First, we developed a basic system using the EDK tool, with RS232 and flash memory hardware, and developed the flash memory control. Next, we attempted to create an ICAP interface for the PowerPC. Finally, we attempted to apply the PlanAhead tool to the EDK design to create reconfigurable modules to test the ICAP module with. Figure 5.1 shows the addition of microprocessor static logic to the top level design from 4.1. It is important to note that the microprocessor is not a black box but has implementation code. Although some of the peripherals may be instantiated as a black box, the top level is not.

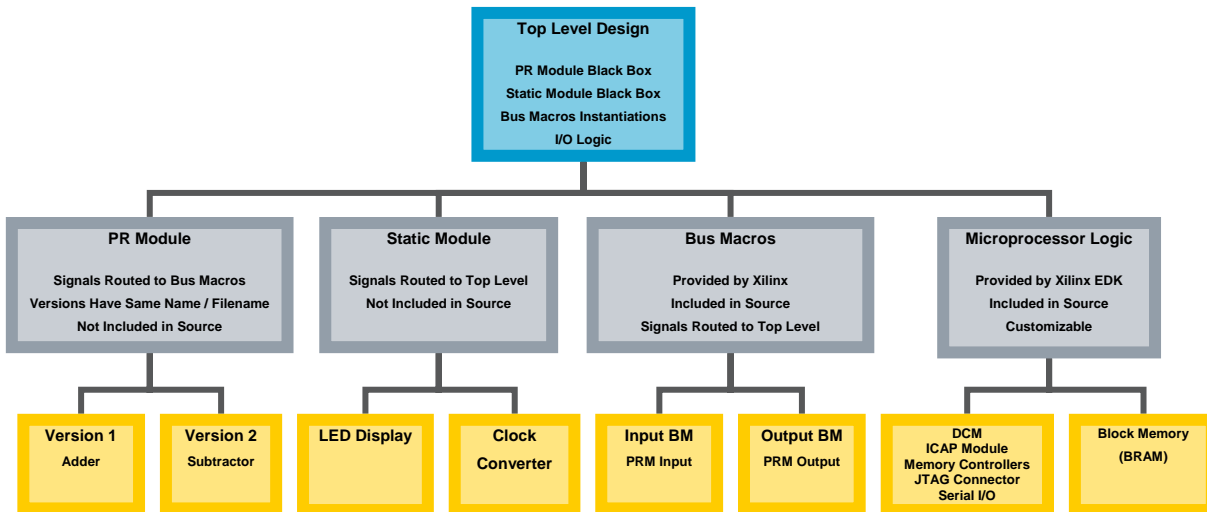


Figure 5.1 – Top Level Design Hierarchy

5.2.2: BRAM vs. Flash Memory

When using the PowerPC and ICAP module to load the reconfiguration data out of memory, as previously discussed, we needed to make a decision between storing the reconfiguration data in flash memory on the P160 Communications Module 2, or in internal block RAM (BRAM). As we decided in the design requirements section, the flash memory has a distinct advantage in the fact that it uses significantly fewer FPGA resources. The flash memory's resource usage is fixed to the size of the memory controller no matter what size memory we address (beyond some additional bits for a larger data or address bus), whereas using BRAM increases resource utilization linearly with the amount of memory required.

Although the flash has the large advantage of not requiring additional FPGA resources, it loses out in that it is significantly more difficult to program. The BRAM can be programmed by simply storing the reconfiguration data in the .DATA section of the corresponding C program, and setting the EDK tools to store the .DATA section in BRAM. The BRAM would therefore be loaded as part of the FPGA's bitstream while programming over JTAG. However, to program the flash memory, a much more complicated method is necessary, as it is not in the JTAG chain and only connected directly to the FPGA itself.

5.2.3: Programming Flash Memory

Programming the flash memory on the expansion communications module is not nearly as simple as programming the platform flash or the FPGA itself (via JTAG), as it is not in the JTAG loop. The expansion flash memory is only accessible via FPGA IO pins, and therefore must be programmed by passing the data through

the FPGA. Xilinx provides a script that executes most standard flash memory commands, which can be accessed in the EDK by using the Device Configuration / Program Flash Memory option. It should be noted, however, that although the script will look like it is running without moving the flashwriter.tcl file to the local project directory, it will only program correctly if done so. The flash memory programming application can program the flash with any binary file, although it is designed for programming ELF or SREC program executables. However, to simplify the design, we will be storing the program itself in BRAM, and simply using the flash memory to store the reconfiguration bitstreams only. The script works by loading a pass-through program using the debugger tool, which passes through the data from the JTAG debugger connection to the physical flash memory, using the FPGA's IO. Figure 5.2, below shows a screenshot of the flash memory programmer GUI.

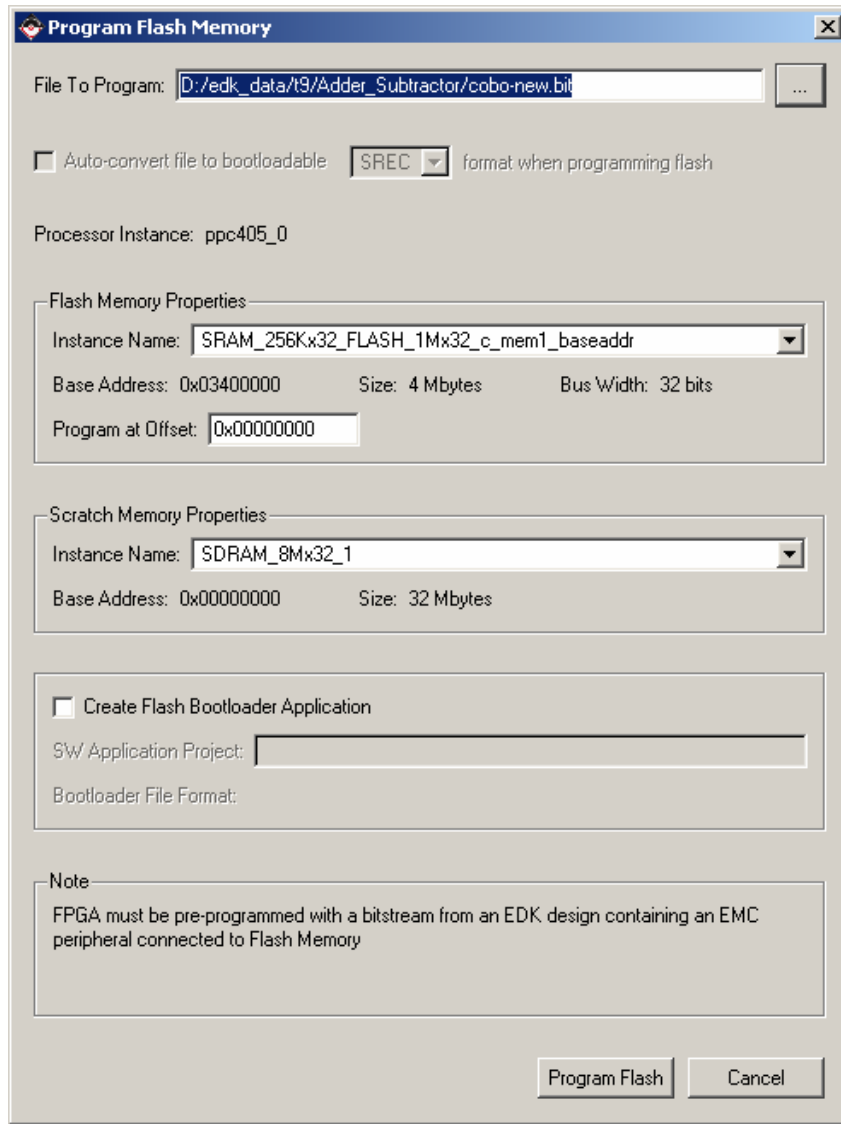
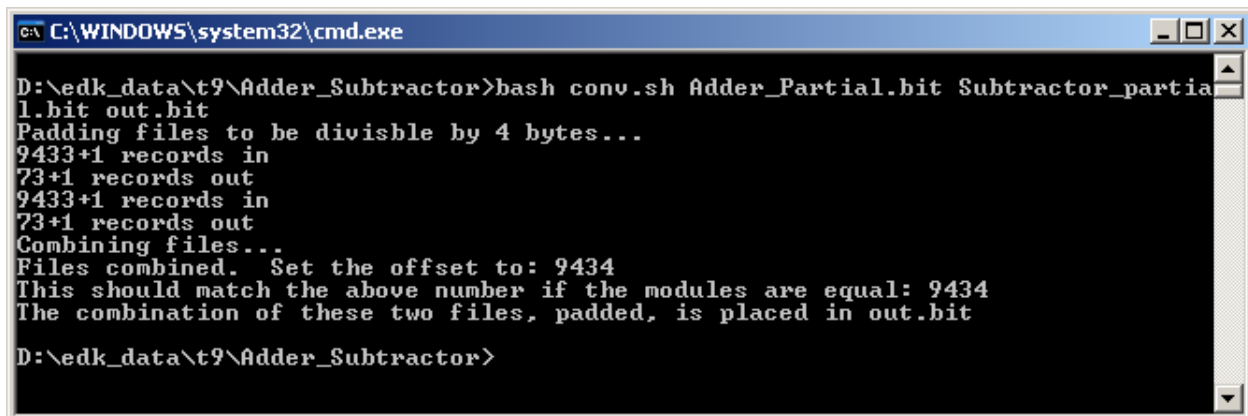


Figure 5.2 – Flash Memory Programmer

Another issue with the flash memory programmer is that it does not support writing multiple files at separate offsets to the flash memory. Although it can program at an offset, it only supports writing one file to that location, and it erases the entire flash before it programs anything, making it useless to store multiple bitfiles. Therefore, a script has been developed which combines multiple bitfiles together, which involves padding them so they are even 32-byte words (rather than single bytes), combing the files together, and then providing the offsets and lengths that are necessary for the PowerPC software to know for programming. An example of the script is shown below, in Figure 5.3.



```
C:\WINDOWS\system32\cmd.exe
D:\edk_data\t9\Adder_Subtractor>bash conv.sh Adder_Partial.bit Subtractor_partia
l.bit out.bit
Padding files to be divisble by 4 bytes...
9433+1 records in
73+1 records out
9433+1 records in
73+1 records out
Combining files...
Files combined. Set the offset to: 9434
This should match the above number if the modules are equal: 9434
The combination of these two files, padded, is placed in out.bit

D:\edk_data\t9\Adder_Subtractor>
```

Figure 5.3 – Flash Combination Script

Once the bit file itself is loaded into flash, it can be accessed by the PowerPC microprocessor by simply addressing its memory location, as with any other memory. This allows very simple portability between using the flash and any other form of memory. The fact that the bit file is in flash memory is completely transparent to the microprocessor code.

5.2.4: Programming - Configuration over ICAP / PowerPC

Utilizing the on-board PowerPC processor to perform self reconfiguration is a slightly more involved process when compared to manual programming over JTAG. Xilinx provides a software development kit known as the Xilinx Platform Studio. The platform studio generates VHDL code to interface the FPGA with the on-board processor. The platform studio also includes an embedded developer's kit and an Eclipse-based interactive development environment which provides the ability to write C and C++ programs that execute on the processor and connected I/O devices. The processor also makes various memory banks available for data segments, code segments, stacks, heaps, and boot loaders. These memory banks come in the form of block RAM (BRAM), static RAM (SRAM), synchronous dynamic RAM (SDRAM), and flash memory.

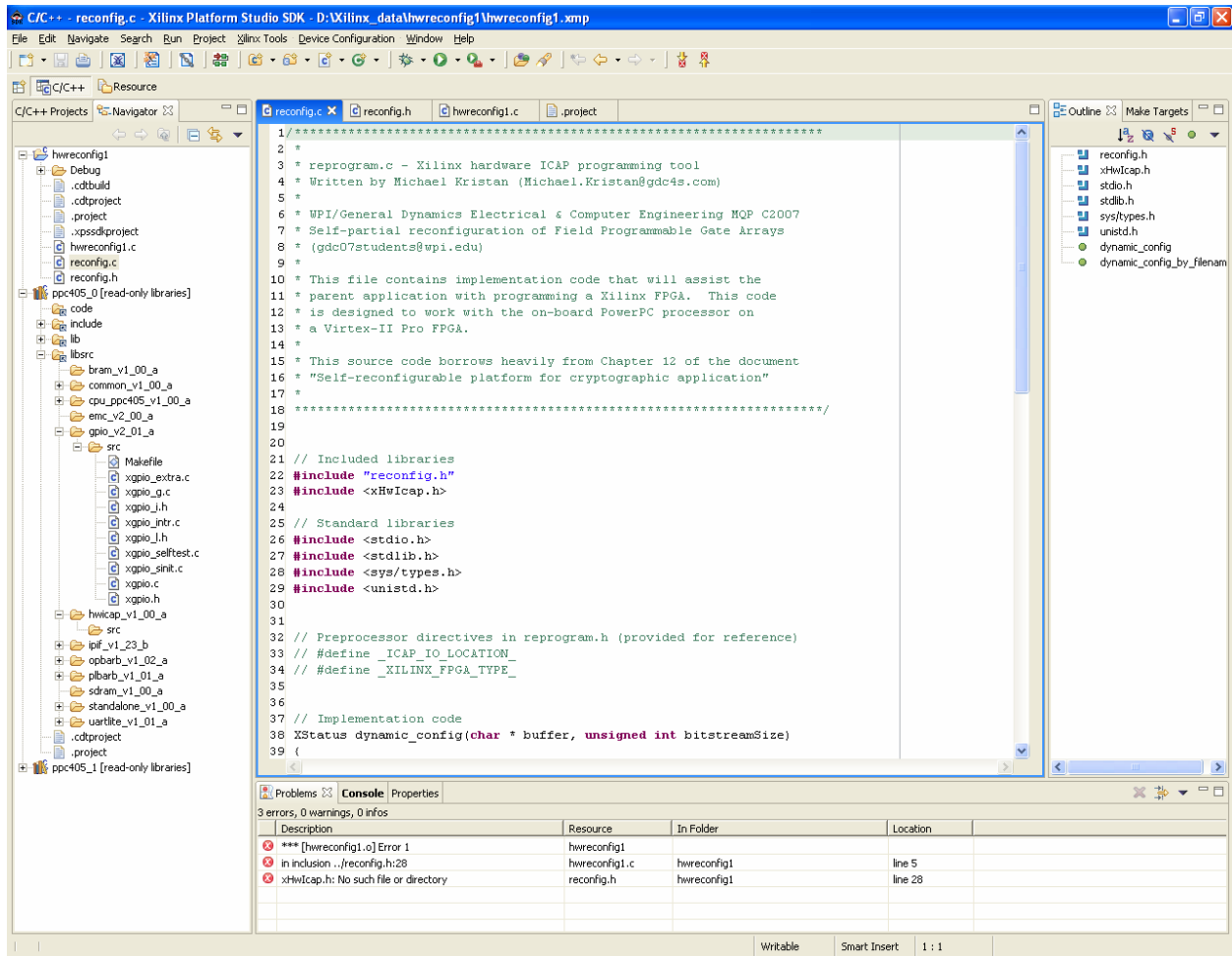


Figure 5.4 – Xilinx SDK Screen Shot

To perform reconfiguration, the embedded C program needs to make use of the Xilinx OPB_HwIcap module and C libraries. By including this library, a programmer can instruct the hardware ICAP module to program a file based on a bitstream that can be referenced by a memory pointer. While this requires the bitstream to be accessible by memory, this is only a minor inconvenience as it can be loaded from the flash at time of programming. This also requires the bitstream to be converted from a binary file to a C standard array. While Xilinx does not provide this feature, third party tools exist that allow this to take place.

5.2.5: PlanAhead Development

Once a hardware system was chosen in the EDK and the hardware is synthesized, the next step was to load the net list into PlanAhead. In PlanAhead, the partial reconfig module was defined and placed in an appropriate PBlock. It is important to note that only one of the actual reconfigurable modules can be done at a time so if that area contains more than one module, a new PlanAhead project is needed each time. Bus macros needed for communication between the reconfigurable module and the static logic had to be placed along with any

external I/O mappings for the bus macros. The rest of the logic is considered to be static and is assigned to the base module PBlock and is not constrained to any specific location. This allows the Xilinx tools to automatically place and route the hardware in any way it sees fit. Upon successful passage of the DRC checks, the floorplan can be exported to a new folder and the PlanAhead partial reconfig flow can be executed. The order of the flow is as follows:

1. Budgeting
2. Static logic implementation
3. Partial reconfig module implementation
4. Final assembly

Once completed, there should be a partial bit stream for the reconfigurable module, a partial bit stream for the blank module, and a full bit stream. These bit files can then be brought back into the EDK so that bit stream initialization can be run for the block memory (BRAM) and the executable files can be loaded.

5.2.6: Flow Integration

The key to successfully running partial reconfiguration while using the PowerPC microprocessor on the Virtex-II Pro chips was to integrate the Embedded Development Kit and Xilinx PlanAhead. A tool flow was created that allowed for development and implementation on the board. The process flow is as follows:

1 Phase 1 – Embedded Development Kit (EDK)

- 1.1 Create a new project in EDK using the Base System Builder
- 1.2 Add peripherals from the IP catalog
- 1.3 Modify appropriate top level logic using the VHDL files (.\hdl folder in EDK project)
- 1.4 Modify synthesis parameters in system_xst.scr to force XST to keep hierarchy
- 1.5 Synthesize using the 'Generate Netlist' command.
- 1.6 Run 'Generate Bitstream' and 'Update Bitstream' in order to initialize BRAM and to compile the embedded program.

NOTE: This process will create its own bit file called 'download.bit', do not use it.

2 Phase 2 - PlanAhead, complete for each unique reconfigurable module

- 2.1 Import all NGC files from the EDK (.\implementation folder in the EDK project) and any defined constraint files into a new PlanAhead project
- 2.2 Draw a PBlock for the reconfigurable modules and assign the reconfig module to that PBlock
- 2.3 Assign all other modules to the base module
- 2.4 Connect all static I/O and bus macros to pins, if appropriate

- 2.5 Run DRC checks
 - 2.6 Export floorplan to a new folder
 - 2.7 Add bus macro files to a subfolder on exported floor plan
 - 2.8 Run PR Budgeting
 - 2.9 Run Static Logic Implementation
 - 2.10 Run Partial Reconfig Module Implementation
 - 2.11 Run Assembly
- 3 Phase 3 – iMPACT**
- 3.1 Assign a new configuration file using the static_full.bit file generated in PlanAhead
 - 3.2 Add system_bd.bmm, the block memory mapping file
 - 3.3 Add executable.elf, the executable code for the
 - 3.4 Program the board

5.3: Issues

5.3.1: Lack of design flow

Flow integration was not a straightforward process as Xilinx does not provide a fully documented and supported method of integrating microprocessor development in the EDK with partial reconfigurable modules in PlanAhead. Xilinx does provide hints that this is feasible and explains how to use PlanAhead to merely generate constraints which the EDK would use for synthesis, mapping, and routing. Xilinx also provided some documentation on how to use the ISE design flow with the EDK but not XST. With the release of Platform Studio 8.2i, ISE has been deprecated and is not recommended for use. Therefore, implementation has been done using XST.

5.3.2: Removal of DCM wrappers

Because the clock needs to be a global signal in a partial reconfiguration implementation, it cannot be contained within a sub module. In order to address this problem, the top-level VHDL file (system.vhd) needed to be modified so that the digital clock module (DCM) was not contained in a black box called a DCM_wrapper. To fix that, the component instantiation had to be removed from the wrapper and placed directly in the top level. This process simply involved copying and pasting the provided code at the higher level.

5.3.3: EDK Makefiles

Because the EDK depends heavily on UNIX makefiles to synthesize hardware and compile code, decoupling the steps in the flow was rather difficult. If files changed in the background, the EDK would automatically remake all the sections it thought needed to be made. This resulted in the synthesis function running many times which took a long time. The makefile also on occasion called the EDK platform generation tool (platgen) which automatically erased customized VHDL files and synthesis parameters and created problems if overlooked. Two customized files that were prone to being overwritten were (based on the EDK project directory root):

- `.\hdl\system.vhd`
- `.\synthesis\system_xst.scr`

5.3.4: Synthesis parameters

One of the benefits of using the ISE Project Navigator to synthesize VHDL code was the ability to specify extra parameters that are needed to successfully do partial reconfiguration. XST however did not provide this in a graphical interface. The synthesis file had to be modified by hand in a text editor to contain the necessary parameters. The complete file can be found in Appendix B.1.

5.3.5: AREA_GROUP errors when mapping bus macros

Xilinx posted a known bug on their website related to running the map command in the partial reconfig tool. Normally, PlanAhead does not automatically copy over the bus macro files when exporting the floorplan. This caused the ngdbuilder to not find the bus macro files. The result is an error message that says that the bus macro does not have an area group assigned to it. To resolve this issue, the bus macro implementation files need to be copied over to the exported floorplan directory as a subfolder and that folder needs to be added to the search path. This can be done by adding the following parameter to the static logic implementation ngdbuild:

`-sd .\<name of subfolder>\`

5.3.6: Memory mapping errors on static implementation

In the static logic implementation, there is an option to add block memory mapping (BMM) files as a parameter for ngdbuild. It made logical sense to add the `system_bd.bmm` auto-generated file that the EDK built. The reasoning is that it would make the router aware of the microprocessor's block memory. The problem was that when the bmm file was added as a parameter, the static logic implementation (step 2.9 in the flow) failed. A screenshot of the error message is shown in figure 5.5.

```
Place & Route Output
Writing NGD file "top.ngd" ...
Writing NGDBUILD log file "top.bld"...

NGDBUILD done.
#-----
# ***** Start map at: 2/23/07 11:22:18 AM *****
#-----
D:\KristanM_snapshot\wpi_hw\Partial_Reconfig_C07\Sources\PlanAhead\hwreconfig2\hwreconfig2exp8-8\static> map -pr b -intstyle ise top.ngd
Using target part "2vp30ff1152-6".
Mapping design into LUTs...
ERROR:MapLib:482 - Blockram rambl6_sl_sl_0 is a memory mapped blockram generated
for the Microprocessor. However it is not connected properly, causing it to
be trimmed. Please connect up all memory mapped blockram properly and re-run
Ngdbuild.
ERROR:MapLib:482 - Blockram rambl6_sl_sl_1 is a memory mapped blockram generated
for the Microprocessor. However it is not connected properly, causing it to
be trimmed. Please connect up all memory mapped blockram properly and re-run
Ngdbuild.
ERROR:MapLib:482 - Blockram rambl6_sl_sl_10 is a memory mapped blockram
generated for the Microprocessor. However it is not connected properly,
causing it to be trimmed. Please connect up all memory mapped blockram
properly and re-run Ngdbuild.
ERROR:MapLib:482 - Blockram rambl6_sl_sl_11 is a memory mapped blockram
generated for the Microprocessor. However it is not connected properly,
causing it to be trimmed. Please connect up all memory mapped blockram
properly and re-run Ngdbuild.
ERROR:MapLib:482 - Blockram rambl6_sl_sl_12 is a memory mapped blockram
generated for the Microprocessor. However it is not connected properly,
causing it to be trimmed. Please connect up all memory mapped blockram
properly and re-run Ngdbuild.
ERROR:MapLib:482 - Blockram rambl6_sl_sl_13 is a memory mapped blockram
generated for the Microprocessor. However it is not connected properly,
causing it to be trimmed. Please connect up all memory mapped blockram
properly and re-run Ngdbuild.
ERROR:MapLib:482 - Blockram rambl6_sl_sl_14 is a memory mapped blockram
generated for the Microprocessor. However it is not connected properly,
causing it to be trimmed. Please connect up all memory mapped blockram
properly and re-run Ngdbuild.
ERROR:MapLib:482 - Blockram rambl6_sl_sl_15 is a memory mapped blockram
generated for the Microprocessor. However it is not connected properly,
```

Figure 5.5 - Map error on Static Implementation

5.4.7: ClearCase check-in errors for binary files

One other observed problem occurred when trying to check in synthesized binary files into the ClearCase repository. ClearCase does not know how to handle files ending in .ut, .ncd, .ngd, & .ngm because it assumes that they are text files. As a result, these files never get checked in. Manual means of transfer (flash drive, network share, email, etc.) are needed to share these files.

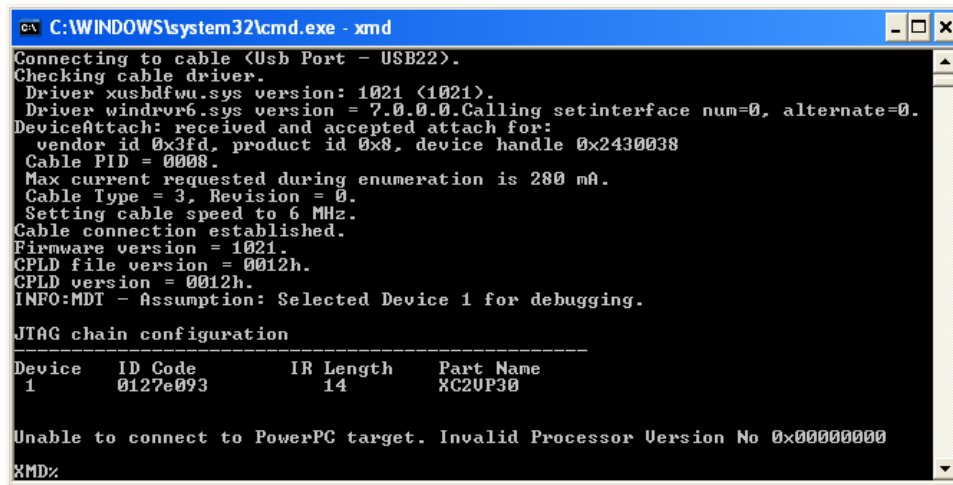
5.4: Results

5.4.1: Successes

After running the partial reconfig tools in PlanAhead, the bit files were successfully loaded onto the FPGA. We were also able to verify that the reconfigurable module can be replaced without disrupting the static logic that was on the board. The Xilinx iMPACT tool was used to load the bit files onto the Virtex-II board.

5.4.2: Failures

The primary failure is that even though the bit files can be successfully loaded and reconfigured while the system is in operation, the microprocessor is not functional. Efforts to make the processor boot or to access any of the peripherals were unsuccessful. As shown in figure 5.6, the JTAG connector cannot reach the PowerPC. The hypothesis is that the PlanAhead tools break the I/O connections to the microprocessor hardware.



```
C:\WINDOWS\system32\cmd.exe - xmd
Connecting to cable (Usb Port - USB22).
Checking cable driver.
Driver xushdfwu.sys version: 1021 (1021).
Driver windrvr6.sys version = 7.0.0.0. Calling setinterface num=0, alternate=0.
DeviceAttach: received and accepted attach for:
  vendor id 0x3fd, product id 0x8, device handle 0x2430038
  Cable PID = 0008.
  Max current requested during enumeration is 280 mA.
  Cable Type = 3, Revision = 0.
  Setting cable speed to 6 MHz.
Cable connection established.
Firmware version = 1021.
CPLD file version = 0012h.
CPLD version = 0012h.
INFO:MDT - Assumption: Selected Device 1 for debugging.

JTAG chain configuration
-----
Device  ID Code      IR Length  Part Name
  1      0127e093         14      XC2VP30

Unable to connect to PowerPC target. Invalid Processor Version No 0x00000000
XMD>
```

Figure 5.6 - PowerPC is unreachable via JTAG and the Xilinx debugger

Conclusion

This project was completed successfully, however due to issues we ran into we were not able to complete all of our original goals. A proof of concept of partial reconfiguration was completed, and work was begun on self-controlled dynamic partial reconfiguration, but not completed. Fortunately, in D term 2007 a group will follow up on this project and attempt to complete the self-controlled dynamic partial reconfiguration hardware.

Goal Completion

At the beginning of this project, three goals were set – to complete a successful proof of concept of dynamic partial reconfiguration, create a proof of concept of self-controlled dynamic partial reconfiguration, and finally setup a storage database which future project teams will have access to. We successfully completed the first and third goal, but ran out of time to complete the second goal due to problems with the software tools used for development.

Our primary goal, a proof of concept of dynamic partial reconfiguration, using the PC and a JTAG tool, was completed successfully. However, this simply proves that partial reconfiguration is possible, and is not particularly useful in any final application, as it requires external control to reconfigure the FPGA. The desired result is to have the reconfiguration logic be internal to the FPGA itself, which was the second goal of the project.

The second goal, of self-controlled dynamic partial reconfiguration, was started but not completed. However, several components critical to the success of this task were completed. A flash module, for reading from the flash on the development board was developed, as well as scripts on the PC side to combine several partial bitstreams together and load them onto the flash memory via the JTAG tool. This was tested and verified to be completed and successful. In addition, software for the PowerPC to interface to the ICAP was developed, however we were not able to test it since the hardware was not successfully completed. The issues preventing the completion of this goal were based on problems integrating the Xilinx PlanAhead and EDK tools.

The third and final goal of the project was to setup a version control system to be used for both this project and future projects. Using the Rational ClearCase tool provided to us by our sponsor, we successfully implemented this. This will allow future MQP project groups and future General Dynamics projects access to our work in a single location. In addition, it provided us a system to keep track of versions of our files while in development.

Future Work

The overall goal the sponsor, General Dynamics C4 Systems, is to develop an encryption system which uses TMR and partial reconfiguration. Our project developed the proof of concept for the partial reconfiguration part, and previous projects have researched the TMR implementation. However, significant work still needs to be completed before the final goal of a working encryption system using PR and TMR can be completed.

Another MQP group will be completing a direct follow-up to our project, in D term 2007. It is our recommendation that they further investigate partial reconfiguration and develop a working system, using self-controlled partial reconfiguration. Beyond that, further groups will need to develop a useable, scalable implementation of TMR before it is feasible to use in an encryption design. In addition, the version of the AES encryption algorithm that the sponsor currently has is far too large to fit three copies on the Virtex 2 FPGA's we were using, and therefore a larger FPGA or more compact encryption hardware must be used. Finally, a dedicated partial reconfiguration toolset (including scripts for linking PlanAhead and EDK) would much improve the scalability and transferability of the work done, and could be investigated as an MQP project if Xilinx does not create one themselves.

References

- [1] AES Questions & Answers, <http://csrc.nist.gov/CryptoToolkit/aes/aesfact.html>
- [2] C. Bobda, A Ahmadinia, K Rajesham, M Majer, "Partial Reconfiguration Design and Implementation Challenges on Xilinx Virtex FPGAs," <http://www12.informatik.uni-erlangen.de/publications/ahmadinia/BARMNo5.pdf>
- [3] C. Kao, "Benefits of Partial Reconfiguration," *Xcell Journal*, vol. 2005, no. 55, pp. 65-69, 2005.
- [4] "Early Access Partial Reconfiguration User Guide," Xilinx PR Early Access Lounge, <http://www.xilinx.com/support/prealounge/protected/docs/ug208.pdf>, March 2006
- [5] Federal Information Processing Standards, "Announcing the Advanced Encryption Standard (AES)," *Federal Register*, vol. 66, no. 40, pp. 12762-3, Feb 2001.
- [6] Federal Register Announcement, <http://csrc.nist.gov/CryptoToolkit/aes/frn-fips197.pdf>
- [7] General Dynamics C4 Systems, <http://www.gdc4s.com/about/>
- [8] G. Braeckman, G. Van de Branden, A Touhafi, G Van Dessel, "Module Based Partial Reconfiguration: a quick tutorial", July 2004.
- [9] G. Mermoud, "A Module-Based Dynamic Partial Reconfiguration Tutorial," <http://ic2.epfl.ch/~gmermoud/files/publications/DPRtutorial.pdf>, Nov 2004.
- [10] H. Bar-El, "Introduction to Side Channel Attacks", Discretix, Israel, 2006.
- [11] J. Lach, W.H. Mangione-Smith, M. Potkonjak, "Low overhead fault-tolerant FPGA systems," *Very Large Scale Integration (VLSI) Systems*, *IEEE Transactions*, vol. 6, no.2, pp. 212-221, Jun 1998.
- [12] K. Parnell, "Could Automotive Processor Obsolescence be History?" *Xilinx*, WP169 (v1.0), San Jose, Oct 2002.
- [13] M. Goosman, N Dorairaj, E. Shiflet, "How to take advantage of partial reconfiguration in FPGA designs," *Programmable Logic Design Line*, <http://www.pldesignline.com>, Feb 2006.
- [14] Memec Design. Virtex-II Pro™ FF1152 Development Board User's Guide. 2005, Avnet, http://avnet.co.jp/products/kits/docs/VirtexIIPro_FF1152_2.pdf
- [15] "Modular Design," Xilinx Toolbox, http://toolbox.xilinx.com/docsan/xilinx8/books/data/docs/dev/dev0025_7.html
- [16] M.P. Lundy, "A Self-Healing Circuit Implementing TMR," Worcester Polytechnic Institute, Worcester, 2006.
- [17] "Partial Reconfiguration," Xilinx Toolbox, http://toolbox.xilinx.com/docsan/xilinx8/books/data/docs/dev/dev0036_8.html

- [18] "Partial Reconfiguration Software User's Guide," Xilinx PR Early Access Lounge, http://www.xilinx.com/support/prealounge/protected/software/pa_pr_user_guide_81.pdf
- [19] "PlanAhead," Xilinx, 2007, http://www.xilinx.com/ise/optional_prod/planahead.htm
- [20] "Platform Studio and the EDK," Xilinx, 2007, http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm
- [21] R.E. Lyons, W. Vanderkulk, "The use of Triple-Modular Redundancy to Improve Computer Reliability," IBM Journal, vol. 1962, pp. 200-209, Apr 1962.
- [22] S. Wichman, S. Adyha, S. Ahrens, R. Ambli, B. Alcorn, Dr. D. Connors, D. Fay, "Partial Reconfiguration Across FPGAs," <http://rogue.colorado.edu/draco/papers/mapldo6-reconfig.pdf>, 2006.
- [23] "Xilinx Applications Notes 290 Two Flows for Partial Reconfiguration," Xilinx, Sep 2004, <http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf>
- [24] "Xilinx History Timeline," Xilinx, 2007, <http://www.xilinx.com/company/history.htm>
- [25] "Xilinx iMPACT," Xilinx, 2007, http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=dr_dt_impact
- [26] "Xilinx ISE Classics," Xilinx, 2007, http://www.xilinx.com/ise/logic_design_prod/classics.htm
- [27] "Xilinx Early Access PR Lounge," Xilinx, 2007, <http://www.xilinx.com/support/prealounge/protected>

Glossary

AES – Advanced Encryption Standard. Made Effective in 2002 by the National Institute of Standards and Technology (NIST) as the main encryption standard in the United States. AES is still widely used across world as of 2007 and is highly secure in comparison with its predecessor the Data Encryption Standard (DES).

ASIC – Application Specific Integrated Circuit; generally faster and less power consuming than FPGAs but lacks the ability to be reprogrammed in the field or be adapted to many different uses.

Bit file – File(s) created by ISE, or PlanAhead which are then directly loaded onto the FPGA. Bit files are the finished product of the design flows and are binary representations used to configure the FPGA.

Bit stream – Typically used to describe the configuration data being loaded onto the FPGA. One or more bit files can be loaded into the bit stream for configuration onto the FPGA.

Black box – Practice of creating blank modules in the top level. Partial reconfigurable designs require that the top level contain no logic. Instead, empty modules are black boxes are created and then referenced to a separate net list. This practice is common in modular design.

BRAM – Block Random Access Memory. BRAM is internal to the FPGA itself and can be referenced directly in embedded applications.

Bus macro – Type of hard macro used to route data to and from reconfigurable modules. Bus macros are required in partially reconfigurable designs to eliminate data loss during reconfiguration. Bus Macros must be physically placed across reconfigurable boundaries within PlanAhead.

Constraints – Term used to describe timing and location requirements of an FPGA design. This includes assignment of I/O pins, Pblock ranges and properties, as well as timing events.

DRC – Design Rule Checks. These tests are run within PlanAhead to check for known design issues and problems. DRCs are completed as an aid to the designer to diagnose problems early on.

Embedded Development Kit (EDK) – Xilinx software toolset used to implement embedded microprocessor applications on Xilinx FPGAs. The EDK provides a GUI for implementation of peripherals available on the board and interfaced with the embedded processors.

Field Programmable Gate Array (FPGA) – A semiconductor device which contains programmable logic blocks. They offer the distinct ability to be programmed and re-programmed after production and are ideal for situations in which designs must be changed often.

Flash Memory – Non volatile memory which is available on the P160 Communications Module. The Flash can be used to store bit files for download to the FPGA and remains stored even when the board is powered down.

Floor-planning – The practice of controlling where modules are physically implemented on an FPGA. Floor-planning is most often used to improve timing and power efficiency in designs by minimizing excess routing. However, floor-planning is required in partial reconfiguration designs for proper creation of partial bit files.

GUI – Graphical User Interface – Term used to describe a software environment in which graphic representation is used in place of plain text. GUIs are generally considered to be more user friendly.

I/O – Term used to describe Input and Output pins.

I/O Buffers – Created at the top level so pins can be properly assigned to ports within the design. Buffers are not desirable at the lower level in modular design and must be turned off.

ICAP – Internal Configuration Access Port. The ICAP can be used internally to control FPGA configuration. In the case of self partial reconfiguration the ICAP is called by the PowerPC to partially reconfigure the FPGA.

JTAG – IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture. The JTAG port is used to interface the FPGA with a PC or other external controller. iMPACT software uses the JTAG connection to complete a boundary scan and load bit files to the FPGA.

LED – Light Emitting Diode. Basic user output on the FF1152 board.

Modular design – Design model considered to be most useful in complex designs created by several team members. Module design follows a specific design flow and is advantageous over basic design flows in that multiple pieces can be designed concurrently and trouble shooting can be done separately. This allows for modules to be changed or repaired without affecting areas which are working properly.

Netlist – File(s) created during synthesis (.ngc, .edif, .edn) which hold the configuration data for implementation of each module or top level design. Net lists are implemented into PlanAhead and then

altered with floor planning data. These files are then used to generate bit files by a number of Xilinx provided scripts.

Pblock – Designation in PlanAhead for Area Groups. Pblocks are drawn in rectangular form for each reconfigurable module. Pblocks are used to reserve physical space for specific modules to be placed routed on the FPGA. A single Pblock is created for all static logic but is not drawn; instead it is left without a range.

PowerPC (PPC) – Microprocessor originally developed by IBM and widely used in Apple personal computers. The PPC has also been implemented in embedded applications such as on the Virtex II Pro and Virtex 4 series FPGAs.

Scrubbing – The practice of rewriting whole or part of an FPGA with code already configured onto it. Scrubbing can be completed at regular intervals increase system reliability.

SDRAM – Synchronous Dynamic Random Access Memory. SDRAM is a type of solid state memory which is most commonly used in Personal Computers. The FF1152 development board provides 64 mb of SDRAM.

Slice – Physical separation of Logic, BRAM, and interconnects on an FPGA. Slice positions are used to define locations of Pblocks. The term slice utilization refers to how large a particular Pblock is in relation to the entire FPGA.

Tool flow – Description of steps required for a successful design. Tool flows vary based on design intent and become more complex with the addition of other tool sets. Common partial reconfiguration designs include the combination if ISE and PlanAhead tool flows while a more complicated self-reconfiguring design requires the addition of the EDK tool flow.

Triple modular redundancy (TMR) – The design practice of replicating each working module three times. A voting circuit is then used to detect output differences as a check to make sure the system has not been compromised. Common TMR systems also replicate the voting circuit to further increase reliability.

Versioned Object Base (VOB) – The name of a source controlled repository in ClearCase. VOBs are the root-level of the repository in which documents and folders can be placed. VOBs can be mounted as a view on a workstation.

VHDL – Also known as VHSIC HDL or Very High Speed Integrated Circuit Hardware Description Language. VHDL is a popular language for the creation of FPGA designs. VHDL code is synthesized in ISE with the Xilinx Synthesis Tool and was the primary design language used in this project. VHDL is different from programming languages as it is actually a description of hardware behavior used to physically create hardware configurations.

Xilinx – Worlds largest developer and original creator of FPGAs. Xilinx makes the Virtex II Pro series FPGA as well as the software tools used in this project.

Xilinx iMPACT – Software used for bit file download to the development board. iMPACT uses a USB or Parallel Platform cable to connect to the FPGA JTAG chain. iMPACT is used when the design is completely implemented and ready to be loaded onto the FPGA.

Xilinx ISE – Base toolset for FPGA design. Includes synthesis and simulation tools and can be used to generate complete basic designs or export hierarchical net lists for use in PlanAhead to create partially reconfigurable designs.

Xilinx PlanAhead – Software used for creation of a partially reconfigurable design. PlanAhead uses exported ISE or EDK net lists and allows for floor-planning as well as constraints creation and design rule checks.

XPS – Xilinx Platform Studio. XPS is the toolset used for the design of embedded applications. XPS is a part of the Embedded Development Kit (EDK).

Appendix A - Basic Circuit – PR over JTAG Design

A.1: Top Level - VHDL

```
-----  
-----  
-----  
-- Worcester Polytechnic Institute  
-- General Dynamics C4 Systems MQP  
-- Michael Kristan  
-- Brian Loveland  
-- Robert Sazanowicz  
-- C Term 2007  
-- Dynamic Partial Reconfiguration of an FPGA  
-----  
-----  
-- Top Level Initial Design for Partial Reconfiguration over JTAG  
-- Synthesize with I/O Buffers and Keep Hierarchy  
-- Modules must remain Black Boxes  
-----  
-----  
  
-- Included Libraries and Packages  
  
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;  
  
-- Top Level Entity Declaration  
  
entity top is  
    Port ( CLOCK      : in    STD_LOGIC;  
          IN1         : in    STD_LOGIC_VECTOR (3 downto 0);  
          IN2         : in    STD_LOGIC_VECTOR (3 downto 0);  
          OUTPUT      : out   STD_LOGIC_VECTOR (3 downto 0);  
          LED1        : out   STD_LOGIC;  
          LED2        : out   STD_LOGIC;  
          LED3        : out   STD_LOGIC;  
          LED4        : out   STD_LOGIC);  
end top;  
  
-- Top Level Behavioral  
  
architecture Behavioral of top is  
  
    -- Declared Components  
  
    component add_module is  
        port ( ADDIN1: in    STD_LOGIC_VECTOR (3 downto 0);  
              ADDIN2: in    STD_LOGIC_VECTOR (3 downto 0);  
              ADDOUT: out   STD_LOGIC_VECTOR (3 downto 0));  
    end component;  
  
    component 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 component;  
  
--Declared Signals
```

```

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

-- Begin Top Level Behavioral

begin

--Instantiate Black Box Modules and Bus Macros

--Reconfigurable Module - Adder Logic

reconfig_module : add_module port map
    (ADDIN1 => sigin1,
     ADDIN2 => sigin2,
     ADDOUT => sigout);

--Static Module - LED Display Logic

base_module: dis_module port map
    (CLK      => CLOCK,
     BLINK    => LED1,
     SOLID    => LED2,
     LIGHT1   => LED3,
     LIGHT2   => LED4);

--Input and Output Bus Macros

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

inputbusleft: busmacro_xc2vp_l2r_async_narrow
    port map(input0  => IN1(0),
             input1  => IN1(1),
             input2  => IN1(2),
             input3  => IN1(3),
             input4  => '0',
             input5  => '0',
             input6  => '0',
             input7  => '0',
             output0 => sigin1(0),
             output1 => sigin1(1),
             output2 => sigin1(2),
             output3 => sigin1(3));

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

inputbusright: busmacro_xc2vp_r2l_async_narrow
    port map(input0  => '0',
             input1  => '0',
             input2  => '0',
             input3  => '0',

```

```

input4      => IN2(0),
input5      => IN2(1),
input6      => IN2(2),
input7      => IN2(3),
output4     => sign2(0),
output5     => sign2(1),
output6     => sign2(2),
output7     => sign2(3);

```

```
end Behavioral;
```

A.2: Reconfigurable Module – Addition Logic - VHDL

```

-----
-----
-----
-- Worcester Polytechnic Institute
-- General Dynamics C4 Systems MQP
-- Michael Kristan
-- Brian Loveland
-- Robert Sazanowicz
-- C Term 2007
-- Dynamic Partial Reconfiguration of an FPGA
-----
-----
-- Reconfigurable Module - Addition Logic for use with Partial Reconfiguration over JTAG Design
-- Synthesize without I/O Buffers and Keep Hierarchy
-----
-----

--Included Libraries and Packages

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

--Entity Declaration

entity add_module is
    port ( ADDIN1: in      STD_LOGIC_VECTOR (3 downto 0);
          ADDIN2: in      STD_LOGIC_VECTOR (3 downto 0);
          ADDOUT: out     STD_LOGIC_VECTOR (3 downto 0));
end add_module;

--Modular Architecture of Component

architecture Modular of add_module is

--Begin Architecture

Begin

    --Addition of two 4 bit inputs resulting in a single 4 bit output

    ADDOUT <= (not(ADDIN1 + ADDIN2));

end modular;

```

A.3: Reconfigurable Module – Subtraction Logic - VHDL

```

-----
-----
-----
-- Worcester Polytechnic Institute

```

```

-- General Dynamics C4 Systems MQP
-- Michael Kristan
-- Brian Loveland
-- Robert Sazanowicz
-- C Term 2007
-- Dynamic Partial Reconfiguration of an FPGA
-----
-----
-----
-- Reconfigurable Module - Subtractor Logic for use with Partial Reconfiguration over JTAG Design
-- Synthesize without I/O Buffers and Keep Heirarchy
-----
-----
-----
-- Included Libraries and Packages

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

-- Entity Declaration

entity add_module is
    port ( ADDIN1: in      STD_LOGIC_VECTOR (3 downto 0);
          ADDIN2: in      STD_LOGIC_VECTOR (3 downto 0);
          ADDOUT: out     STD_LOGIC_VECTOR (3 downto 0));
end add_module;

-- Modular Architecture of Component

architecture Modular of add_module is

-- Begin Architecture

Begin

    -- Subtracts 2nd 4 bit input from 1st 4 bit input resulting in a 4 bit output
    ADDOUT <= (ADDIN1 - ADDIN2);

END modular;

```

A.4: Static Module – LED Display – VHDL

```

-----
-----
-----
-- Worcester Polytechnic Institute
-- General Dynamics C4 Systems MQP
-- Michael Kristan
-- Brian Loveland
-- Robert Sazanowicz
-- C Term 2007
-- Dynamic Partial Reconfiguration of an FPGA
-----
-----
-----
-- Static Display Module for use with Adder-Subtractor PR over JTAG Design
-- Synthesize without I/O Buffers and Keep Heirarchy
-----
-----
-----
--Included Libraries and Packages

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

```

```

use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Entity Declaration

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;

-- Modular Architecture

architecture Modular of dis_module is

    --Declared Components

    component Clk_Convrt is
        Port (Clk_in      : in   std_logic;
              Reset      : in   std_logic;
              Clk_1Hz,
              Clk_10Hz,
              Clk_10KHz  : out  std_logic);
    end component;

--Begin Architecutre of Display Module

begin

    -- Clock Conversion Component - Creates 2 Hz, 20 Hz, and 20Khz Clocks (With XC2VP30-
    FF1152)
    -- Borrowed From WPI ECE 3801 Online Laboratory Resources
    -- Note: Code was initially designed for use with a Spartan 3 board and a 50 MHz Internal
    Clock
    -- The FF1152 board used with this design includes a 100 Mhz clock, therefore the clock
    signals
    -- mentioned are actually doubled in speed. This is irrelevant as the frequency is not
    important
    -- to the design and is only used as a visual testing aid for PR implementation

    clockconverter : Clk_Convrt port map ( Clk_in      => CLK,
                                           Reset       => '0',
                                           Clk_1Hz     => BLINK,
                                           Clk_10KHz  => SOLID);

    -- LEDs ON - Tie Remaining LEDs to Logic 0 (FF1152 LEDs operate on Inverted Logic)

    LIGHT1 <= '0';
    LIGHT2 <= '0';

end Modular;

```

A.5: Clock Converter Module

```

-----
-----
-----
-- This Code is borrowed from the ECE 3801 Online Laboratory Resources
-- Initially designed for use with a 50 Mhz clock, the actual clock speeds are off
-- by a factor of two. This module is only used as a testing aid to ensure PR design
-- is free of glitches. 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;

-- This code generates three clock signals
-- (1 Hz, 10 Hz, 10 KHz) from a single
-- 50 MHz clock provided by the Spartan3 board

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

```

Appendix B – Self Reconfiguration – PR over ICAP design

B.1: XST synthesis parameter file (system_xst.scr)

```
run
-keep_hierarchy YES
-rtlview Yes
-glob_opt AllClockNets
-read_cores YES
-write_timing_constraints NO
-cross_clock_analysis NO
-bus_delimiter <>
-case maintain
-slice_utilization_ratio 100
-verilog2001 YES
-fsm_extract YES -fsm_encoding Auto
-safe_implementation No
-fsm_style lut
-ram_extract Yes
-ram_style Auto
-rom_extract Yes
-mux_style Auto
-decoder_extract YES
-priority_extract YES
-shreg_extract YES
-shift_extract YES
-xor_collapse YES
-rom_style Auto
-mux_extract YES
-resource_sharing YES
-mult_style auto
-bufg 16
-register_duplication YES
-register_balancing No
-slice_packing YES
-optimize_primitives NO
-tristate2logic Yes
-use_clock_enable Yes
-use_sync_set Yes
-use_sync_reset Yes
-iob auto
-equivalent_register_removal YES
-slice_utilization_ratio_maxmargin 5
-opt_mode speed
-opt_level 1
-p xc2vp30ff1152-6
-top system
-ifmt MIXED
-ifn system_xst.prj
-ofn ../implementation/system.ngc
-hierarchy_separator /
-iobuf YES
-max_fanout 10000
-sd {../implementation}
```

B.2: Sample ICAP test program

```
/*****
 *
 * hwreconfig2_1.c - Xilinx hardware ICAP sample application
 *
 *****/
```



```

* WPI/General Dynamics Electrical & Computer Engineering MQP C2007
* Self-partial reconfiguration of Field Programmable Gate Arrays
* (gdc07students@wpi.edu)
*
* This file contains implementation code tests to see if partial
* reconfiguration works using the on-board ICAP module. This code
* is designed to work with the on-board PowerPC processor on
* a Virtex-II Pro FPGA.
*
*****/

// Included libraries
#include "xparameters.h"
#include <stdio.h>
#include "xutil.h"
#include "reconfig.h"

#include "bitstream.dat" // A bit file converted to a C array

int main (void)
{
    int bitstream_length;
    bitstream_length = 1000;

    print("Greetings!\r\n");

    xil_printf("WPI/General Dynamics C07 MQP\r\n");
    xil_printf("gd07students@wpi.edu\r\n");
    xil_printf("Self partial reconfiguration utility\r\n");

    //xil_printf("Enter length of the bitstream : ");
    //scanf("%d", &bitstream_length);

    xil_printf("The number you entered is : %d", bitstream_length);

    xil_printf("\nTesting partial reconfiguration...\r\n");

    if(dynamic_config(bitstream_loc, bitstream_length) != XST_SUCCESS)
    {
        print("Failure");
    }
    else
    {
        print("Success");
    }

    xil_printf("\nExiting main ()\r\n");
    return 0;
}

```

B.3: HWICAP User-defined tools – Header file

```

/*****
*
* reprogram.h - Xilinx hardware ICAP programming tool
*
* WPI/General Dynamics Electrical & Computer Engineering MQP C2007
* Self-partial reconfiguration of Field Programmable Gate Arrays
* (gdc07students@wpi.edu)
*
* This file contains function prototypes 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.
*
*****/

```

```

// Pre-processor directives

// Declaration of local variable to prevent multiple includes
#ifndef REPROGRAM_H_
#define REPROGRAM_H_

// Constant declarations
#define _ICAP_IO_LOCATION_ 0xffff80000
#define _XILINX_FPGA_TYPE_ XHI_XC2VP30

// External libraries
#include "xHwIcap.h"

// Function prototypes
XStatus dynamic_config(char * buffer, unsigned int bitstreamSize);

#endif /*REPROGRAM_H_*/

```

B.4: HWICAP User-defined tools – Implementation code

```

/*****
 *
 * reprogram.c - Xilinx hardware ICAP programming tool
 *
 * WPI/General Dynamics Electrical & Computer Engineering MQP C2007
 * Self-partial reconfiguration of Field Programmable Gate Arrays
 * (gdc07students@wpi.edu)
 *
 * 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 borrows heavily from Chapter 12 of the document
 * "Self-reconfigurable platform for cryptographic application"
 *
 *****/

// Included libraries
#include "reconfig.h"
#include "xHwIcap.h"

// Standard libraries
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

// Preprocessor directives in reprogram.h (provided for reference)
// #define _ICAP_IO_LOCATION_
// #define _XILINX_FPGA_TYPE_

// Implementation code
XStatus dynamic_config(char * buffer, unsigned int bitstreamSize)
{
    // Declarations of local variables
    Xuint32 * current_data;
    *current_data = (Xuint32)buffer;
    XHwIcap * MyHWICAP = _ICAP_IO_LOCATION_;

    // Initialize Hardware ICAP
    XStatus HWICAP_status;
    HWICAP_status = XHwIcap_Initialize(&MyHWICAP, 0, _XILINX_FPGA_TYPE_);
    if(HWICAP_status != XST_SUCCESS)

```

```

        {
            return HWICAP_status;
        }
    XHwIcap_CommandDesync(&MyHWICAP);

    // Program the FPGA
    HWICAP_status = XHwIcap_SetConfiguration(&MyHWICAP, (Xuint32*)current_data,
(Xuint32)(bitstreamSize/4));
    return HWICAP_status;
}

XStatus dynamic_config_by_filename(char * filename)
{
    // Declarations of local variables
    FILE *fp;
    unsigned int fileSize;
    char * buffer;

    // Open file
    fp = fopen(filename, "rb");
    if(fp == NULL)
    {
        exit(1);
    }

    // Get size of file
    fseek(fp,0,SEEK_END);
    fileSize = ftell(fp);
    rewind(fp);

    // Allocate buffer
    buffer = (char*)malloc(fileSize);
    if(buffer == NULL)
    {
        exit(2);
    }

    // Copy file contents into buffer
    fread(buffer, 1, fileSize, fp);

    // Call dynamic_config
    return dynamic_config(buffer, fileSize);
}

```

B.5: Bit file to array Perl tool

```

#
# bin2hex.pl by Chami.com
# http://www.chami.com/tips/
#
# Modified by Michael Kristan
# Michael.Kristan@gdc4s.com
#
# February 2007
# Fixed small errors when sending out
# to C-style arrays
#
# For academic/research purposes ONLY
#

# number of characters per line
$chars_per_line = 15;

# -----

```

```

# perl bin2hex.pl <binary_file_name> <language_id>
# language id
#
# 0 = Perl (default)
# 1 = C / C++
# 2 = Pascal / Delphi
#
$lang = $ARGV[1];

$rem_begin = "begin binary data:";
$rem_end   = "end binary data.";

# initialize for Perl strings
# by default
#$_var      = "# $rem_beginn".
            "$bin_data = # %dn";
#$_begin    = "";
#$_end      = "";n";
#$_break    = ".n";
#$_format   = "\x%02X";
#$_separator = "";
#$_comment  = "# $rem_end ".
            "size = %d bytes";

# C / C++
if(1 == $lang)
{
    $_var      = "/* $rem_begin */".
                "char bin_data[] = ".
                "/* %d */";
    $_begin    = "{";
    $_end      = "}";
    $_break    = "";
    $_format   = "0x%02X";
    $_separator = ",";
    $_comment  = "/* $rem_end ".
                "size = %d bytes */";
}
elseif(2 == $lang)
{
    $_var      = "{ $rem_begin }n".
                "const bin_data : ".
                "array [1..%d] of ".
                "byte = n";
    $_begin    = "(";
    $_end      = ");n";
    $_break    = "n";
    $_format   = "%02X";
    $_separator = ",";
    $_comment  = "{ $rem_end ".
                "size = %d bytes }";
}
}

if(open(F, "<". $ARGV[0]))
{
    binmode(F);

    $s = '';
    $i = 0;
    $count = 0;
    $first = 1;
    $s .= $_begin;
    while(!eof(F))
    {
        if($i >= $chars_per_line)
        {
            $s .= $_break;
            $i = 0;
        }
        if(!$first)

```

```

    {
        $s .= $_separator;
    }
    $s .= sprintf(
        $_format, ord(getc(F)));
    ++$i;
    ++$count;
    $first = 0;
}
$s .= $_end;
$s .= sprintf $_comment, $count;
$s .= "";

$s = ".sprintf($_var, $count)$.s;

print $s;

close( F );
}
else
{
    print
    "bin2hex.pl by Chami.comn".
    "n".
    "usage:n".
    " perl bin2hex.pl <binary file>".
    " <language id>n".
    "n".
    " <binary file> : path to the ".
    "binary filen".
    " <language id> : 0 = Perl, ".
    "1 = C/C++/Java, ".
    "2 = Pascal/Delphin".
    "n";
}

```

B.7: Bitstream Combiner Script

```

# run script as ./conv.sh infile1.bit infile2.bit outfile.bit
echo "Padding files to be divisble by 4 bytes..."
dd if=$1 of=$1-e ibs=4 conv=sync
dd if=$2 of=$2-e ibs=4 conv=sync
echo "Combining files..."
cat $1-e $2-e > $3
size1=$(cat $1-e | wc -c)
calcsize=$(expr $size1 / 4)
echo "Files combined. Set the offset to: $calcsize"
size2=$(cat $2-e | wc -c)
calcsize=$(expr $size2 / 4)
echo "This should match the above number if the modules are equal: $calcsize"
rm -rf $1-e
rm -rf $2-e
echo "The combination of these two files, padded, is placed in $3"

```

B.8: Diff on customized system.vhd based on generated file

```

33,40c33,40
<     sys_rst_pin : in std_logic;
<         IN1 : in  STD_LOGIC_VECTOR (3 downto 0);
<         IN2 : in  STD_LOGIC_VECTOR (3 downto 0);
<         OUTPUT : out STD_LOGIC_VECTOR (3 downto 0);

```

```

< LED1 : out STD_LOGIC;
< LED2 : out STD_LOGIC;
< LED3 : out STD_LOGIC;
< LED4 : out STD_LOGIC
---
> sys_rst_pin : in std_logic
>
>
>
>
>
>
46,47d45
< -----
< -- Declared Components for PR top level
49,110c47,110
< component busmacro_xc2vp_l2r_async_narrow is
< port (
< input0 : in std_logic;
< input1 : in std_logic;
< input2 : in std_logic;
< input3 : in std_logic;
< input4 : in std_logic;
< input5 : in std_logic;
< input6 : in std_logic;
< input7 : in std_logic;
< output0 : out std_logic;
< output1 : out std_logic;
< output2 : out std_logic;
< output3 : out std_logic;
< output4 : out std_logic;
< output5 : out std_logic;
< output6 : out std_logic;
< output7 : out std_logic
< );
< end component;
<
< component busmacro_xc2vp_r2l_async_narrow is
< port (
< input0 : in std_logic;
< input1 : in std_logic;
< input2 : in std_logic;
< input3 : in std_logic;
< input4 : in std_logic;
< input5 : in std_logic;
< input6 : in std_logic;
< input7 : in std_logic;
< output0 : out std_logic;
< output1 : out std_logic;
< output2 : out std_logic;
< output3 : out std_logic;
< output4 : out std_logic;
< output5 : out std_logic;
< output6 : out std_logic;
< output7 : out std_logic
< );
< end component;
<
< component add_module is
< port ( ADDIN1: in STD_LOGIC_VECTOR (3 downto 0);
< ADDIN2: in STD_LOGIC_VECTOR (3 downto 0);
< ADDOUT: out STD_LOGIC_VECTOR (3 downto 0));
< end component;
<
< component 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 component;
<
<
<
<     signal sigin1, sigin2, sigout: std_logic_vector (3 downto 0);
<     --signal clocktemp: std_logic;
< -----
< ---
< >
< >
< >
1038c1038
<     component dcm_0_module is
< ---
< >     component dcm_0_wrapper is
1062c1062
<     -- attribute box_type of dcm_0_wrapper: component is "black_box";
< ---
< >     attribute box_type of dcm_0_wrapper: component is "black_box";
1312,1313d1311
< -----
< -- Added for PR reconfig design
1315,1401d1312
<     reconfig_module : add_module port map          (ADDIN1 => sigin1,
<
<         ADDIN2 => sigin2,
<
<         ADDOUT => sigout);
<
<
<     base_module: dis_module port map              (CLK => sys_clk_pin,
<
<         BLINK => LED1,
<
<         SOLID => LED2,
<
<         LIGHT1 => LED3,
<
<         LIGHT2 => 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');
< --
< --         clk0 => CLOCK,
< --         ce0 => '1',
< --         ce1 => '1',
< --         ce2 => '1',
< --         ce3 => '1',
< --         clk1 => '0',
< --         clk2 => '0',
< --         clk3 => '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');
< --                                clk0 => CLOCK,
< --                                ce0 => '1',
< --                                ce1 => '1',
< --                                ce2 => '1',
< --                                ce3 => '1',
< --                                clk1 => '0',
< --                                clk2 => '0',
< --                                clk3 => '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');
< --                                clk0 => CLOCK,
< --                                ce0 => '1',
< --                                ce1 => '1',
< --                                ce2 => '1',
< --                                ce3 => '1',
< --                                clk1 => '0',
< --                                clk2 => '0',
< --                                clk3 => '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));
< --                                clk0 => CLOCK,
< --                                ce0 => '1',
< --                                ce1 => '1',
< --                                ce2 => '1',
< --                                ce3 => '1',
< --                                clk1 => '0',
< --                                clk2 => '0',
< --                                clk3 => '0');
1403c1314,1403
< -----
<
>
2313c2313
<   dcm_0 : dcm_0_module
<
>   dcm_0 : dcm_0_wrapper

```


Appendix C: Basic PR – Troubleshooting Guide

Q1. Why is the Partial Reconfiguration Flow Tool shown in grey in PlanAhead?

Partial reconfiguration is not initially enabled in the PlanAhead software. To use it, you must first enable the following script.

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

Q2. Why are all IO ports marked as “IBUF” or “OBUF” and why am I unable to constrain them in PlanAhead?

This error results from incorrect synthesis option within the ISE or EDK. You must leave IO Buffers on during top level synthesis and turn them off during module level synthesis. When this is done properly, you will be able to properly constrain input and output ports in the PlanAhead package view.

Q3. How do I place the bus macros within PlanAhead?

Bus macros must be placed manually in the device view. To do this you must first turn Site Constraint Mode on in the top right toolbar. Then, find the bus macros in your primitive net list view. You can then drag each bus macro into the device view placing them by moving the mouse over the lower left and corner of the slice to the left of the reconfigurable module boundary. Remember, bus macros are unidirectional so the function of each macro as an input or output is dependant on which direction macro you are using and what side of the module it is placed on. A properly placed macro will straddle the reconfigurable module and span across two full slices on the FPGA. DRC checks will alert you if the macro is placed incorrectly. Xilinx recommends keeping input macros on the left side of and output macros on the right side of each reconfigurable module. This may simplify the amount of routing required.

Q4. DRC Checks fail and claim that the reconfigurable rectangles are not properly located.

Although all Pblocks must be drawn with a rectangle, PlanAhead may claim that the rectangle is incorrectly placed. To do this, be sure that the rectangle is large enough for the logic confined within it. To do this, check the attributes for the module and be sure that the available values are larger than required. Second, you must place the minimum x and y values for the reconfigurable module on an even slice, and the maximum values on an odd slice. To check this, select the properties window for the module and double click on the rectangles tab.

Q5. Why do the design rule checks return an error of type “TPSYNC for asynchronous bus macro”?

When using asynchronous bus macros this error will always be returned and can be ignored if asynchronous macros are sufficient for your design. PlanAhead recommends the use of synchronous

macros to avoid problems in timing critical paths. A solution to this is shown in the Design Rule Violation window.

Q6. The Partial Re-config Tool Flow is not completing correctly, but DRC checks all pass.

Check to make sure that the floor plan was exported properly to a new and empty directory. Also, remember that each step in the partial reconfig flow must be completed in order. This is also required if any changes are made. For example, if static budgeting is rerun, all steps following it must also be rerun. This is because PlanAhead simply calls scripts which edit existing files. In some cases, the files are copied to a new location before changed and in other cases they are not. Running a step out of order may corrupt files needed for subsequent steps.

Q7. I can assign modules to their appropriate pblocks but I still receive an error when attempting to run the initial budgeting and static implementation phases.

Check to make sure that the appropriate net lists were added in the initial project creation. PlanAhead will show the modules accordingly because the instantiations are present in the top level design even if their separate net lists were not added. Select the File menu and choose the update net list option. Make sure that each module has the appropriate net list assigned to it rather than the top level list.

Q8. PlanAhead is unable to locate any of the required files in the Assemble Stage.

As of PlanAhead 8.2.7 you must remember to export the floor plan to new or empty directory. If files exist within the directory, PlanAhead will create a .data directory inside of it. This causes an error in the directory path when scripts are called later in the process.

Q9. I've completed my design in PlanAhead but the Assemble stage will not complete correctly as it is unable to locate one or more .ncd files.

If the directory path is correct, and the files do exist, this can be explained by a bug within the PlanAhead software. To work around this, simply run the Budgeting and Implementation steps in GUI mode. Then, generate a script file for the Assemble stage instead. When the script is run, this error should not occur and if your design is correct, it will complete successfully. This bug was not fixed as of PlanAhead 8.2.7.

Q10. The partial Re-config tool was run and completed successfully, what next?

The tool should have created bit files for download to the FPGA. Locate the export folder you created on disk. The \merge directory inside of this folder will include 3 bit files for each reconfigurable design. They will be labeled with static_full, pblock_*cv_routed_partial, and pblock_*_blank. Note that the asterisk

denotes the name you chose for the reconfigurable pblock. Remember that you need to create a new PlanAhead project for each version of the reconfigurable design.

Q11. How do I create a design with multiple reconfigurable modules?

A separate PlanAhead project must be created for each version of a reconfigurable module. Once the initial project is completed, simply start a new one. When importing the initial net lists, choose a different reconfigurable version than in the previous project. (NOTE: You must only choose one version for each project). Then, import the constraints file from the exported directory from the previous project. This constraints file will apply all the changes required in PlanAhead saving you a lot of time and work. Next, run the Partial Re-config Tool as before remembering to first export the floor-plan to a new empty directory.

Appendix D: EDK Troubleshooting Guide

Q1. Why does the flashwriter script fail on identifying the flash type?

Be sure that you have downloaded the bitstream for the current project to the device. The flashwriter script relies on the addresses of the project you are working on to run correctly.

Q2. Why does flashwriter execute but the flash doesn't get programmed?

The flashwriter script must be in the project directory that it is executing in, otherwise it won't correctly program the flash. It will look like it is executing, but because of a path problem, it will not execute correctly.

Q3. Why does the bitstream combiner script fail?

You must have GNU tools (bash, dd, expr, rm and cat specifically) in your path for this to work correctly.

Q4. Why doesn't the code execute when I choose program hardware?

Check your linker script – if you want the PowerPC to boot your code, without using the XMD debugger over JTAG, everything except for dynamic memory (stack and heap) must be in BRAM.

Q5. Why don't standard stdio functions work?

Xilinx provides proprietary stdio functions. For example, printf is xil_printf.

Q6. Why can't I edit the source code within Xilinx Platform Studio?

There are two types of XPS projects, elf-only projects and full projects. If the source code is edited in Xilinx Platform Studio SDK (the eclipse-based software development environment), it must be built in SDK as well, and can not be modified in XPS.

Q7. Why won't my program boot upon download?

Check that the option "Mark to initialize BRAM" when right-clicking on the software project is checked off. Also, verify that the processor has the correct executable .elf file in the boot option.

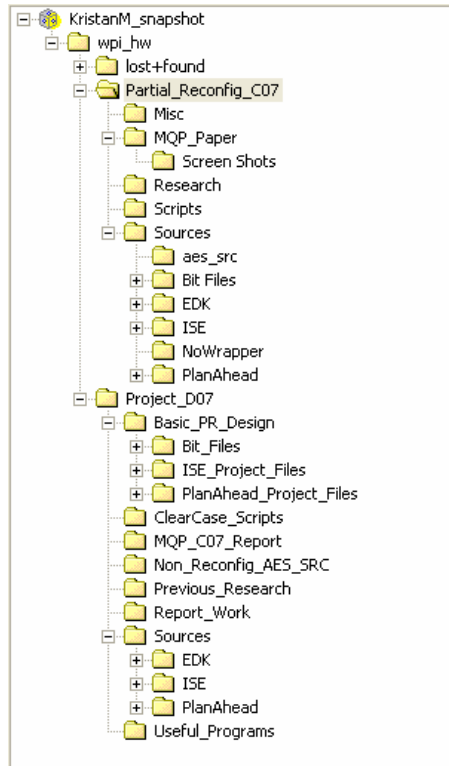
Q8. Why do I get read-only errors when trying to compile, generate netlists, or bitstreams?

If you're working in a ClearCase dynamic view, you cannot do any of those. Either switch to a snapshot view or do EDK project work in a non-ClearCase directory.

Appendix E: ClearCase information

E.1: VOB directory structure

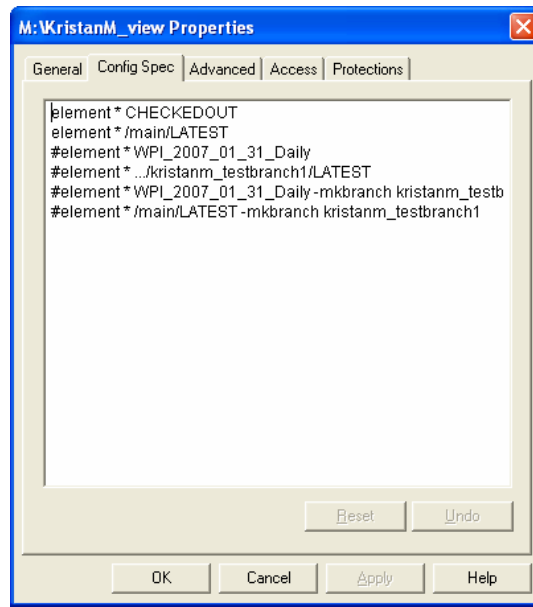
The directory structure of the WP I VOB is shown in the screenshot below:



Project files are found within the sources folder. EDK projects were placed in the EDK folder, ISE projects were placed in the ISE folder, and so on.

E.2: ClearCase config spec

Because our team decided not to do branching, all branch related entries were commented out with a # sign. Changing the order of the items (ClearCase evaluates config specs from top to bottom) changed the way the view displayed items. To revert to a previous day's snapshot, we uncommented the daily label and moved it above the /main/LATEST entry. A sample screenshot of a working config spec is provided.



A view's config spec can be accessed by right clicking on the view's icon in Rational ClearCase explorer on the left vertical bar and selecting view properties.

E.3: Add to source control recursive script (add-to-src-control.bat)

```
@rem= 'PERL for Windows NT -- cperl must be in search path
@echo off
cperl %0 %1 %2 %3 %4 %5 %6 %7 %8 %9
goto endofperl
@rem '
```

```
#####
# Begin of Perl section
#
# Written by IBM Corporation
#
# Adapted & modified by Michael Kristan
# Michael.Kristan@gdc4s.com
```

```
$start_dir = $ARGV[0];

#
# Fixed variable
#
$S = "\\ ";
$list_file = "c:". $S. "list_file";
$list_add = "c:". $S. "list_add";
$choosed = "c:". $S. "choosed";
$max_elements = 150; #To prevent clearcase from freaking out and crashing the video card
```

```
sub clean_file
{
$status = system("del $list_file > NUL 2> NUL");
$status = system("del $list_add > NUL 2> NUL");
$status = system("del $choosed > NUL 2> NUL");
}
```

```

#
# Start of the script...
#

$counter = 0;          # Simple loop counter

printf("add-to-src-control $start_dir...\n");

clean_file();
$status = system("cleartool ls -view_only -r -s $start_dir > $list_file");
open(LIST_ELEMENT,$list_file);
while (($element=<LIST_ELEMENT>) && ($counter < $max_elements) )
{
    $counter++;

    chop $element;
    # printf " Processing $element ";
    if ($element =~ /CHECKEDOUT/)
    {
        # printf(" checkedout file \n");
    }
    else
    {
        # printf " view private \n";
        printf " Processing $element ...\n";

        #
        # For files with spaces...
        #
        if ($element =~ / /)
        {
            $status = system("cmd /c echo \"$element\" >> $list_add");
        }
        else
        {
            $status = system("cmd /c echo $element >> $list_add");
        }
    }
}
close(LIST_ELEMENT);

if (-e $list_add)
{
    $listelement = `type $list_add`;
    $listelement2 = `type $list_add`;
    $listelement2 =~ s/\n//g;
    $listelement =~ s/\n/,/g;
    $status = `echo $listelement > $list_add`;

    $status = system("echo $listelement2");

    # $status = system("clearprompt list -outfile $choosed -dfile $list_add -choices
    # -prompt \"Choose element(s) to put over version control : \" -prefer_gui");

    if ($status != 0)
    {
        # printf("\n Aborting ...\n");
        clean_file();
        exit $status;
    }

    #
    $listtoadd = `type $choosed`;
    $listtoadd =~ s/\n//g;
    #used to be $listotadd
    printf("\n cleardlg /addtosrc $listelement2");
    $status = system("cleardlg /addtosrc $listelement2");

    clean_file();
    exit $status;
}

```

```

}
else
{
# printf("\n No files founded...\n");
clean_file();
exit $status;
}

# End of Perl section

__END__
:endofperl

```

E.4: Timestamp label generator (datelabel.sh)

datelabel.sh ran every day from hostname kristanm.gdc4s.com as a scheduled task. This scheduled task ran each business day at 12:15PM. The purpose was to allow the group to take a daily snapshot of the VOB should there be a need to revert a view to a previous date. Bash does need to be installed in order to run this script. To make running this script easier, a .bat wrapper was created with the simple command **bash ./datelabel.sh**.

```

#!/bin/bash

# Label generator for ClearCase VOB
# Written by Michael Kristan (Michael.Kristan@gdc4s.com)
# Internal Use Only
# This script lives in the \wpi_hw\Partial_Reconfig_C07\Scripts directory

# Generate the label using the bash date command

CLEARCASE_DATE_LABEL=$(date +"WPI_"%Y_"_%m"_"_%d"_"_Daily")
echo "Daily label generation for WPI_HW ClearCase VOB"
echo "Generating label: $CLEARCASE_DATE_LABEL"

# Change folders to the VOB root and generate label

cd ../../
cleartool mklbtype -nc $CLEARCASE_DATE_LABEL
cleartool mklabel -replace -recurse -version \\main\\LATEST $CLEARCASE_DATE_LABEL .

```

E.5: Miscellaneous recursive commands

Each of the batch files in this section is called from within the cleartool find command. The find command is run in the base directory which the recursion starts from. An example command would be:

```
cleartool find . -exec "cmd /c revert.bat"
```

E.5.1: Revert checked out files to the VOB version (revert.bat)

```

@echo off
cleartool uncheckout -rm "%CLEARCSE_PN%"
cleartool checkout -nwarn -unr -nc "%CLEARCASE_PN%"

```


E.5.2: Push changes to VOB but keep elements checked out (push.bat)

```
@echo off
cleartool checkin -nc -nwarn "%CLEARCASE_PN%"
cmd /c cleartool checkout -nwarn -unr -nc "%CLEARCASE_PN%"
```

E.5.3: Recursive unreserved check out

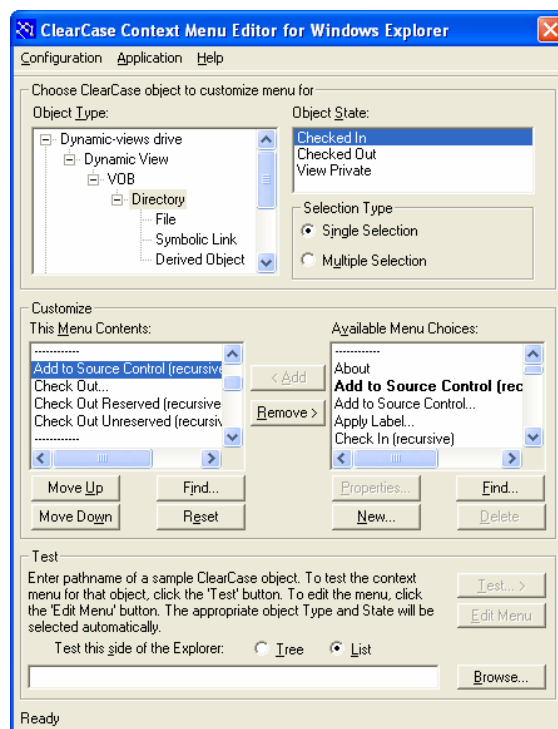
```
cleartool checkout -unr -nc "%CLEARCASE_PN%"
```

E.5.4: Recursive check in

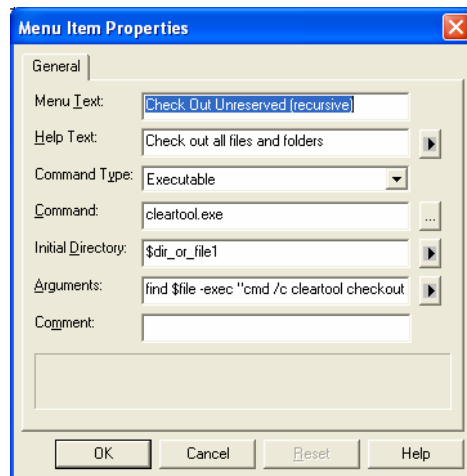
```
cleartool checkout -unr -nc "%CLEARCASE_PN%"
```

E.6: ClearCase Menu Administrator

A useful trick was to add the previously mentioned clear case scripts to the right click menu within Windows Explorer. ClearCase provides a program called “ClearCase Context Menu Editor” to do that. To access it, go to Start, click on Run, and type in **clarmenuadmin** and click ok. The following windows should appear:



From within the menu editor, new menu choices can be added for each of the scripts such as a recursive unreserved check out.



New menu choices appear on the right hand side in “Available Menu Choices” and then can be added to an appropriate menu context. An example would be selecting a dynamic view directory as an object type with an object state of checked in.

E.7: Snapshot views vs. Dynamic views

ClearCase views come in two different types, snapshot and dynamic. Both types of views allow users to check out files, make changes, and check them back in. Dynamic views operate similarly to a network drive, all of the files are stored on the server in read only form (if you check out a file, it is no longer read only). Snapshot views copy the entire VOB to the local hard drive.

General Dynamics strongly suggests using dynamic views because dynamic views automatically reflect changes that other users check-in just like a traditional network drive whereas snapshot views require manual updates. The only advantages offered by snapshot views are that they are faster because the data is stored on the local hard drive and that files can be hijacked and/or deleted by the Xilinx tools during synthesis or build phases. Deleting files in a dynamic view require use of the **cleartool rname** and **cleartool rmem** commands.