Using Relocatable Bitstreams For Fault Tolerance

THESIS

David P. Montminy, Captain, USAF

AFIT/GCE/ENG/07-09

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/07-09

Using Relocatable Bitstreams For Fault Tolerance

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

David P. Montminy, B.S.E.E.

Captain, USAF

March 2007

AFIT/GCE/ENG/07-09

Using Relocatable Bitstreams For Fault Tolerance

David Montminy, B.S.E.E.

Captain, USAF

Approved:

[illegible]	20 Feb 07
Dr. Rusty Baldwin (Chairman)	date
Barry Mullins	20 Feb 07
Dr. Barry Mullins (Member)	date
[illegible]	20 Feb 07
Maj Paul Williams, PhD (Member)	date

AFIT/GCE/ENG/07-09

Abstract

This research develops a method for relocating reconfigurable modules on the Virtex-II (Pro) family of Field Programmable Gate Arrays (FPGAs). A bitstream translation program is developed which correctly changes the location of a partial bitstream that implements a module on the FPGA. To take advantage of relocatable modules, three fault-tolerance circuit designs are developed and tested. This circuit can operate through a fault by efficiently removing the faulty module and replacing it with a relocated module without faults. The FPGA can recover from faults at a known location, without the need for external intervention using an embedded fault recovery system. The recovery system uses an internal PowerPC to relocate the modules and reprogram the FPGA. Due to the limited architecture of the target FPGA and Xilinx tool errors, an FPGA with automatic fault recovery could not be demonstrated. However, the various components needed to do this type of recovery have been implemented and demonstrated individually.

Acknowledgements

First and foremost, without the loving support of my wife this research would not have been possible. I am very grateful to Dr. Rusty Baldwin for giving me the freedom and support to explore and develop applications for partial configuration as I conceived them while keeping me headed in the right direction. Maj Paul Williams provided a great sounding board for new ideas and helped me develop troubleshooting techniques whenever it looked like I had reached a dead end. I also like to thank the international partial reconfiguration community, especially Dr. John Williams of the University of Queensland and Xilinx Research Labs.

David P. Montminy

Table of Contents

Page

List of Figures

List of Tables

List of Abbreviations

Using Relocatable Bitstreams For Fault Tolerance

I. Introduction

1.1 Overview

The modern military, in fact modern society, has become reliant on complex electronic systems, such as satellites, which must provide reliable service. Designing these system to be fault tolerant means these systems can continue operating even if a fault occurs. Methods to incorporate fault tolerance in Field Programmable Gate Arrays (FPGAs) include implementing redundancy and reprogramming the FPGA to recover from a fault.

1.2 Motivation and Goals

The use of FPGAs in operational systems continues to grow as their capabilities increase. Using FPGAs in space and military applications require them to be highly dependable and reliable. Partial reconfiguration can be used to make FPGAs fault tolerant, increasing their dependability and availability, by allowing an FPGA to restore its functionality after a fault has been detected. Traditionally, fault tolerance has been achieved through redundancy, implementing critical systems multiple times.

The goal of this research is to develop a more efficient method for implementing a fault tolerant system on an FPGA, based on bitstream relocation, which implements additional redundancy only when needed.

1.3 Organization

Chapter II provides an introduction to fault tolerance, FPGAs, and using partial reconfiguration to implement fault tolerance. Chapter III describes three fault tolerant configurations that use relocatable modules and describes the software used to perform bitstream relocation. Chapter IV explains the challenges of implementing a dynamic reconfiguration system on the target FPGA. Chapter V presents the conclusions of this study and suggestions for future research.

II. Literature Review

2.1 Introduction

Field programmable gate arrays are digital integrated circuits that can be programmed and reprogrammed post-fabrication by a user to implement a custom circuit. As they have evolved, the size, complexity and computational power of FPGAs have increased making FPGAs not only a valuable tool for rapid prototyping and testing, but also for implementing actual production systems.

The submicron scale of FPGAs increased the number of transistors on each device making them more powerful. As the transistor size has been reduced, the current density in the devices has increased making them more vulnerable to gamma particle radiation [LMSP99]. Changes in the state of a transistor as a result of radiation is called is a Single Event Upset (SEU). Two types of SEUs, soft and hard can be caused by a charged particle. Non-destructive soft errors appear as transient pulses in logic or bitflips when they occur in memory. Single hard-errors (SHEs) are potentially destructive, causing a permanent change in the operation of the device. SHEs include Single Event Latchups, gate rupture, and frozen bits [NAS00].

2.1.1 Applications of Fault Tolerance. Spacecraft engineers design to minimize power, weight, volume and cost while increasing functionality. Many of the components that provide these characteristics, including FPGAs, are susceptible to SEUs [NAS00]. Terrestrial devices can be repaired in place, but intervention in space applications is usually too expensive or impossible. With their increased use in critical scientific and military systems with high reliability requirements, fault-tolerance techniques to improve the reliability and dependability of FPGAs must advance as well.

2.1.2 Motivation for using FPGA reconfiguration for Fault-Tolerance. Fault tolerance has traditionally been provided by building redundancy into a design. In FPGAs, designs have been hardened by replicating components and using techniques such as Triple Modular Redundancy. However, since the area within an FPGA is limited, replication is an expensive approach. An alternative is to provide fault tolerance through dynamic reprogramming of the FPGA.

2.2 Fault Tolerance

Fault-tolerance is one way of providing a dependable computing system. A dependable system provides a quality of service that can justifiably be relied upon. A system's service is the expected behavior of a system as perceived by a user or other systems [Lap85]. A system *failure* is caused by an *error* which results in a system response not in compliance with its expected service. An error in a component or the design of a system is called a *fault* [AL81].

Faults can be classified by their duration, nature and extent [Nel90]. A fault's duration is transient, intermittent or permanent. Transient faults are nonrecurrent, are typically caused by external forces, and are manifested for a finite amount of time. Intermittent faults can cause the system to cycle between faulty and error-free operation. Hard faults are permanent and can be the result of a defect in the design of a component or physical damage. The extent of a fault can be measured by the number of components affected. A local fault affects a single component, while a global fault affects multiple components.

System dependability can be achieved through the use of one or more methods which can be classified into four categories [Lap85]:

- **Fault-avoidance:** Prevents fault occurrence by construction,
- **Fault-tolerance:** Provides, by redundancy, service complying with the system requirement despite faults,
- **Error-removal:** Minimizes, through verification, the presence of latent errors, and
- **Estimating:** by evaluation, the presence, the creation and the consequences of errors.

A fault tolerant circuit continues to provide dependable results even if a fault occurs during operation. In an environment where multiple faults can be expected such as space applications, systems may be required to tolerate multiple faults before the system malfunctions [KZJS00].

There are four phases of fault-tolerance [AL81].

1. **Error detection:** The manifestation of a fault, that is the errors it causes, must be detected so action can be taken.
2. **Damage confinement and assessment:** Since there will likely be a delay between when a fault occurs and when an error is detected, the state of the system must be evaluated to determine if the error has spread within the system. Unless the error is confined, it could cause errors throughout the system.
3. **Error recovery:** To continue operation, the system must be restored to an error-free state.
4. **Fault treatment and continued service:** Once the system has been restored to an error-free state, steps must be taken to enable the system to resume providing the service required by its specification.

The design of a system determines the complexity of performing each of these phases. The relationship between a fault and the error that results can be complex and careful consideration by the designer is needed to isolate faults. Although error detection is usually the starting point for fault tolerance, the other three phases can occur in any order. Decisions made during design can eliminate the need for one or more of the fault-tolerant phases [AL81].

The effectiveness of a fault tolerant system is measured by its reliability and availability. Reliability can be measured by evaluating how a circuit functions when a fault is introduced to the circuit [KZJS00]. Availability is determined by the time needed to restore the circuit to proper operation. Reliability and availability are key measures of dependability.

More formally, reliability, is [Nel90]

$$R_{system} = P(no\ fault) + P(correct\ operation|fault) \times P(fault). \qquad (2.1)$$

Reliability is a function of how faults affect the system and what mechanisms are in place to prevent system failure when a fault occurs [Nel90].

For systems where maintenance cannot be performed and dependable service must be provided for a long period of time, reliability must be high. Reliability can be achieved in two ways. First, the probability that a system does not have a fault can be increased by using higher quality components and other fault avoidance design techniques [Nel90]. Alternatively, the system can be designed to recover from a fault when one occurs. For systems where maintenance cannot be performed such as space systems, mechanisms must be in place so the system can repair itself.

For systems that can be repaired or can perform their own repairs, availability is a useful measure of dependability. Availability is the probability a system is operational at time t. In steady state, availability is the probability that a system will be operational at any random time. Availability can also be expressed as the amount of downtime over a specified interval. Availability can be increased by increasing the expected time between system failures or by reducing the expected amount of time to restore a failed system to operation [Nel90].

2.2.1 Methods for Fault Tolerance. There are a number of methods that can improve a system's fault tolerance. Fault tolerant strategies typically include one or more of the following [Nel90]:

- **Masking:** Correction of generated errors,
- **Detection:** Detection of an error or the manifestation of a fault,
- **Containment:** Preventing an error from propagating across boundaries,
- **Diagnosis:** Identification of the faulty module causing the error,

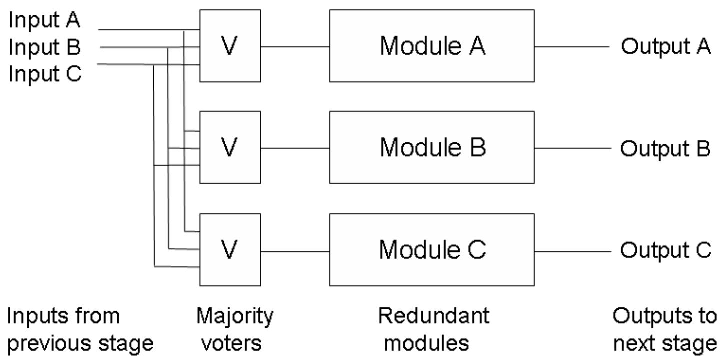


Figure 2.1: One Stage of a TMR Circuit [Nel90].

- **Repair/Reconfiguration:** Repairing, replacing, or bypassing the module, and
- **Recovery:** Restoring the system to a stable state to allow continued operation.

The cost and complexity of implementing these techniques is highly dependent on the system they are implemented in. Combinations of these strategies can be implemented through hardware, software, information and time redundancy [AL81].

Hardware redundancy uses additional hardware to detect or tolerate faults. There are three types of hardware redundancy: passive, active, and hybrid [McF94]. For systems that cannot afford down-time associated with repairing faults, static or passive techniques allow a system to mask some number of faults [Nel90]. Active redundancy techniques detect faults and take action to correct faults. Hybrid redundancy combine masking to prevent fault propagation with fault detection and recovery to remove the faulty module from the system [McF94].

One common form of passive redundancy is triple modular redundancy (TMR). In TMR, three modules compute the same function and the three results are sent to a voter which chooses the majority result. To prevent the voter from being a single point of failure, three voters can be used at each stage. Figure 2.1 illustrates one stage of a TMR circuit. TMR can be generalized to have more than three modules. This is called N-modular redundancy (NMR) and can tolerate up to $\lfloor (N-2)/2 \rfloor$ module failures.

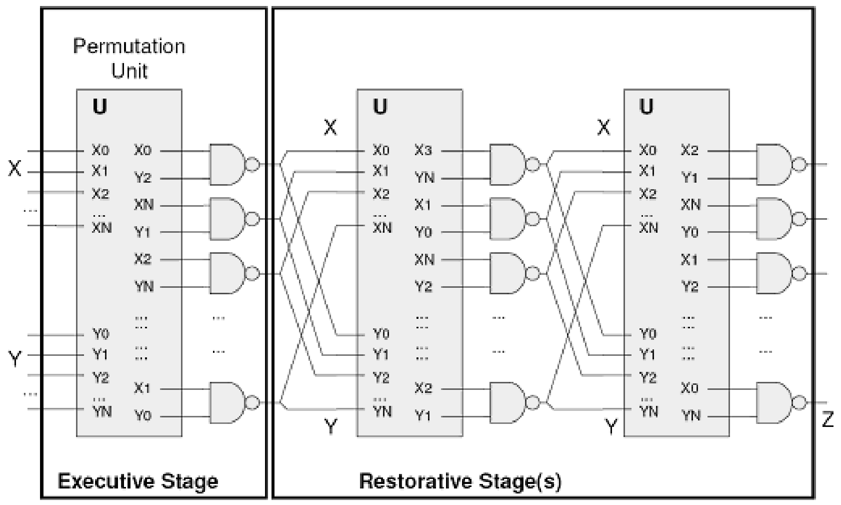


Figure 2.2: NAND Multiplexer [VN56].

NAND multiplexing can be used to reliably perform the Boolean NAND function in the presence of errors that would change the output of a single NAND gate [VN56]. A NAND multiplexer performs the NAND operation redundantly, increasing the probability of a correct result. As shown in Figure 2.2, a NAND multiplexer is comprised of an executive stage and one or more restorative stages. The restorative stage consists of two executive stages in series. Each executive stage has a row of NAND gates in parallel and a permutation unit which determines which input signals will serve as inputs to each NAND gate. If there are no errors, all of the outputs are identical. When all outputs are not identical, a threshold for the number of matching outputs determines the correct result.

In dynamic redundancy, faulty components are detected, diagnosed, and or repaired or replaced [Nel90]. Dynamic redundancy methods typically switch module and/or reconfigure communication routes as faults occur. The location, use and number of spares differentiate active redundancy techniques. These techniques provide diagnosis and repair, but do not mask the fault.

Hybrid redundancy combines passive and dynamic techniques to provide redundancy which both mask faults and repairs the circuit. N-modular redundancy with K-standby sparing uses N-modules which each perform the same function. As with

TMR and NMR, a voter or group of voters determines the correct result. If one of the N-modules fails, it is replaced by one of K-spares.

2.2.2 Reconfiguration as a Method for Fault Tolerance. Since modern FPGAs can be configured to reprogram themselves, they make excellent platforms for dynamic and hybrid redundancy systems. Furthermore, passive redundancy techniques such as NMR or NAND multiplexing although expensive to implement using traditional hardware, can mask faults while a module is replaced or bypassed using reconfiguration techniques.

A number of techniques have been proposed for reconfiguration, including logic block replacement via rerouting, reconfiguring entire columns or rows and shifting entire circuits by row or column within an FPGA to avoid a fault cell in a column [DP94]. A number of techniques are described in Section 2.4, but the specific FPGA architecture and reprogramming methods are important considerations for each method since the viability of each technique is a function of the architecture of the target FPGA.

2.3 FPGAs

In 1984, Xilinx introduced a new class of integrated circuits called the field programmable gate array, or FPGA. The basic FPGA is an integrated circuit consisting of logic blocks, interconnects, and I/O blocks. Logic blocks can be individually configured to perform various functions and are connected using programmable interconnects. Figure 2.3 shows the basic structure of an FPGA. An FPGA configuration, including the function each logic block implements and its connections, is determined when the FPGA is programmed. This programmable architecture means FPGAs are highly configurable with fast design and modification times. In addition, the large number of logic cells, embedded RAM blocks, embedded multipliers, and adders that make up today's FPGAs means they can implement large and complex functions [Max04].

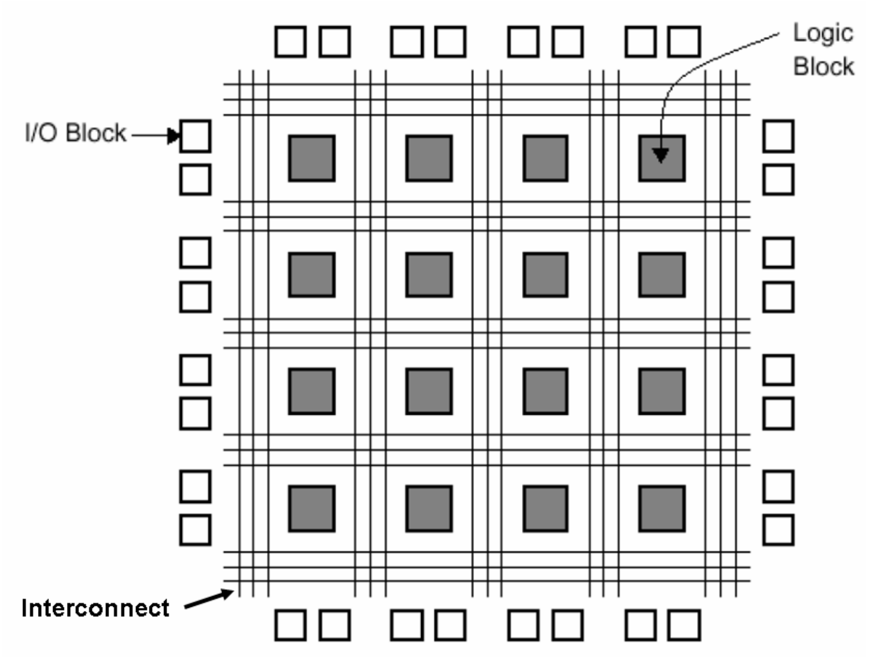


Figure 2.3: The Basic Structure of an FPGA [Kha02].

The design of the logic blocks vary between types of FPGAs and FPGA manufacturers. The granularity of an FPGA refers to the complexity of the logic blocks. If a logic block can support a simple function, the FPGA is considered to be fine-grained. A coarse-grained FPGA architecture can implement a more complex function. Since each coarse-grained logic block has more logic in it, the number of logic blocks needed to implement a function is less than the number of logic blocks needed in fine-grained FPGAs. Although fine-grained FPGAs usually implement a function more efficiently (since most resources in the logic blocks are used), a larger number of interconnects are needed to connect the finer-grained logic blocks.

In Xilinx FPGAs, logic blocks are known as logic cells (LC). Although the exact configuration of logic cells differ between device families, each contains a *3*, *4* or *6*-input look up table (LUT), which can be configured as a 16 x 1 RAM or a 16-bit shift register, a multiplexer and a register as well as other logic [Max04, Xil07b]. Figure 2.4 shows how the basic elements are arranged within the LC. Logic functions are implemented using the LUT. An n-input LUT is programmed to return the result of a logic function based on the values of its n-inputs. A multiplexer can select the

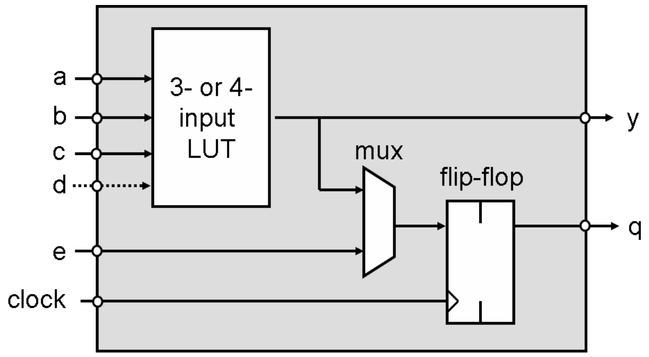


Figure 2.4: Key Elements of a Xilinx FPGA Logic Block [Max04].

result from the LUT or from an input external to the LC. The register acts as a flip-flop or a latch.

In the Virtex-II Pro, logic cells are grouped into slices [Xil05b]. Each slice has two logic blocks and each configuration logic block (CLB) is made up of four vertical slices. There are different types of connections between FPGA resources. The connections lengths vary from connecting CLBs to their neighbors to the connecting to the backbone routing network 24 horizontal and vertical long line routing resources that span the full height and width of the device [Xil05b].

The function an FPGA performs is determined by the programmer and must be programmed into the FPGA. There are two basic types of programmable FPGAs: anti-fused based and memory-based.

Anti-fuse FPGAs are one-time programmable devices with special connections that start as high resistance open circuits but become low resistance connections when programmed. Anti-fuse based FPGAs are non-volatile. They retain their configuration even when power is removed. Additionally, the anti-fuse interconnect structure is relatively immune to the effects of radiation so its configuration not changed as a result of a SEU. However, other transistors on the device remain susceptible to SEUs, so designs and radiation hardening must still be used for anti-fuse devices in high radiation environments [Max04].

Memory-based FPGAs can be reprogrammed in the field, giving them added flexibility, thus making them a viable platform for testing reconfigurable fault-tolerance methods. Two currently available memory-based FPGAs are static random access memory (SRAM) and electronically erasable programmable read-only memory-based (EEPROM). SRAM FPGAs use SRAM configuration cells that can be reconfigured over and over. The function of the circuit is based entirely on the contents of the configuration memory.

EEPROM-based FPGAs have the advantage of being non-volatile. Once they have been programmed they retain their programming even when the device is powered down. Although they can be reprogrammed in the field, their programming time is typically three times longer than similar SRAM devices [Max04].

State of the art FPGAs have multiple embedded microprocessors known as microprocessor cores. A hard microprocessor core is implemented in a pre-defined and dedicated area within the FPGA. Xilinx offers FPGAs with one, two, or four PowerPC 405 hard microprocessor cores. These cores are embedded directly into the FPGA fabric and can be connected to user defined circuitry. A soft microprocessor core is a group of programmable logic blocks configured as a microprocessor. Soft cores are slower and simpler than hard cores, but as many soft core processors as space allows can be placed onto an FPGA. Soft microprocessor cores are available as intellectual property (IP) from FPGA vendors. The MicroBlaze^{TM} processor is a classic 32-bit processor designed specifically to work with the hardware features of Xilinx FPGA devices [WB04].

Designs for the MicroBlaze^{TM} processor can be obtained as intellectual property from Xilinx. For FPGAs, intellectual property, commonly referred to as IP, can be developed by the FPGA vender or a third party. Available IP cores implement a wide array of functions, from signal processing to I/O interfaces. There are three types of IP cores offering different levels of abstraction and flexibility: soft, firm and hard cores [KJdlTR05]. Soft cores are hardware independent synthesizable Hardware

Description Language (HDL) descriptions with a high level of flexibility since they can be extensively modified. Firm cores are technology independent netlists that offer some flexibility through customizable parameters. Hard cores are preplaced and have fixed routing limiting their flexibility. Although a number of IP designs can be used without licensing, typically a fee must be paid to use the designs.

2.3.1 SRAM FPGA Technology. The basic structure of Xilinx SRAM-based FPGAs is a two-dimensional array of logic blocks are linked with vertical and horizontal programmable interconnect channels [Tor02]. The configuration of SRAM FPGAs is controlled by memory cells that are volatile and must be configured each time the FPGA is powered-up. All aspects of the user design are implemented by the configuration memory including LUT equations, signal routing, BlockRAM (BRAM) configuration and BRAM interconnects [Xil05b]. A bitstream provides the configuration control commands and configuration data to the FPGA. On Virtex-II Pro devices, the bitstream is delivered through a serial, boundary scan, or SelectMAP interfaces. The bitstreams for each interface are, for the most part, identical [Xil05b].

Serial mode programs an FPGA using a serial programmable read-only memory device. In serial configuration mode, the bitstream is clocked into the FPGA one bit at a time. In master serial mode, the FPGA drives the clock. Slave serial configuration mode allows FPGAs to be configured by another device such as a microprocessor or master FPGA, with the other device controlling the clock of the slave FPGA [Xil05b].

The JTAG interface (named for the Joint Test Action Group (JTAG) responsible for development of the IEEE 1149.1 standard), the Test Access Port (TAP) and Boundary Scan Architecture, allows in-system programming. Boundary scans allow devices and internal circuitry to be tested. The boundary-scan TAP is used to serially apply tests which can detect opens and shorts at the board and device level. Many vendors have added vendor specific instructions to their boundary-scan implementation allowing configuration instructions [Xil05b]. Using a special instruction, the FPGA can connect the internal SRAM configuration shift registers to the

JTAG scan chain, allowing the FPGA to be programmed using the data-in pin of its TAP [Max04]. The Virtex-II Pro can also be reconfigured using the JTAG port by applying an appropriate partial bitstream through the TAP [Xil05b]. Since partial bitstreams are used to implement specific circuits at specific locations within the FPGA, partial bitstreams used for dynamic reconfiguration are also referred to as hard cores, or more generally cores, as described in Section 2.3.

For internal access to the FPGA configuration and read back operations, Virtex II-Pro devices include an internal configuration access port (ICAP). The ICAP provides access to the FPGA configuration memory using the SelectMAP interface with an 8-bit bidirectional data bus to the Virtex-II Pro configuration logic [Xil05b]. Xilinx developed an on-chip peripheral bus (OPB) core to interface to the ICAP. The HWICAP core allows an embedded processor to read and write the FPGA configuration bitstream through the ICAP at runtime one frame at a time [Xil04c]. A frame is the smallest unit that can be reprogrammed on the Virtex-II Pro and is 1-bit wide slice of a column as shown in Figure 2.5. Devices in the Virtex II-Pro family have between 22 and 94 CLB columns and 22 frames per column [Xil05b]. Software programs running on the embedded processor can construct a custom bitstream by modifying the frames sent to the HWICAP and thereby modify an FPGA circuit as desired at runtime.

2.3.2 SRAM FPGA Reconfiguration. Reconfiguration is the act of reprogramming an SRAM FPGA without resetting, or powering down the device. Reconfiguration can be performed on the whole device *(full reconfiguration)* or a portion of it *(partial reconfiguration)*. The device can be put into the shutdown state for reprogramming or the device can continue to operate, a process known as *active reconfiguration*. Full reconfiguration reinitializes memory contents, while the content of data memory is preserved during partial reconfiguration. In Virtex II-Pro devices partial reconfiguration is only possible through the JTAG and SelectMAP interfaces which includes the ICAP [Xil05b].

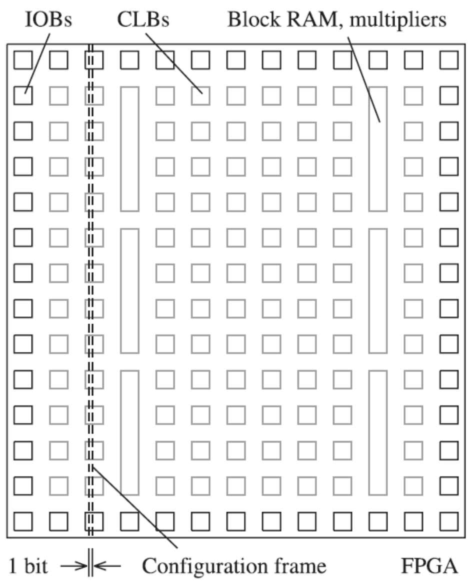


Figure 2.5: Architecture of the Virtex-II FPGA [SBB+06]. In the Virtex-II series the CLBs, BRAM and multiplier are organized in columns. The I/O blocks are arranged around the perimeter of the FPGA. A configuration frame spans the entire column programming a fraction of one CLB or BRAM column and a portion of I/O blocks above and below the columns.

Dynamic partial reconfiguration partially reconfigures an active array while the active circuits not being changed continue to function. *Self-reconfiguration* is a form of dynamic reconfiguration in which specific circuits within an FPGA control the reconfiguration of other portions of the FPGA. With such a dependency, the proper operation of the reconfiguration circuitry must be ensured before, during and after reconfiguration.

SRAM FPGA devices rely on an external configuration control interface to boot and program the FPGA with an initial configuration when power is first applied or the device is reset. Once initially configured via some external method, self-reconfiguration uses an interface within the FPGA driven by internal FPGA circuitry, which may include a microprocessor. The ICAP provides this functionality on Virtex-II, Virtex-II Pro, and Virtex-4 devices. That is, the ICAP enables self-reconfiguration without external hardware [BJRK+03].

Eliminating external circuitry through self-reconfiguration minimizes the latency of accessing an external configuration port. It also minimizes the distance between the control logic and the reconfiguration control logic within the same logic array. System complexity is also reduced, since fewer discrete devices are required [BJRK$^+$03].

Although FPGAs are designed with a regular structure and every logic block can be connected to another logic block, finding a path from one block to another is not a trivial task. Because FPGAs, have a limited number of interconnections, not all connections are possible for a given configuration. Algorithms to reconfigure the FPGA must know the architecture of the FPGA, and which resources are being used, for the algorithm to construct an appropriate reconfiguration bitstream. Although quick reconfiguration, which improves availability, is a goal of fault-tolerant systems, efficient utilization of the FPGA's resources must also be considered. Reconfiguration algorithms should minimize the FPGA resources used by the circuit after reconfiguration and the time to perform the reconfiguration.

To compare the performance of fault-tolerance schemes, benchmark designs can be used. The Microelectronic Center of North Carolina (MCNC) and ISPD98 benchmark suites are commonly used by developers of automated design systems to compare and validate their designs [Alp98]. Reconfiguration changes the layout of the user circuit within the FPGA. By comparing the performance of benchmarks circuits before and after fault-tolerance techniques have been applied, the effectiveness of different techniques can be evaluated. Hardware resources and the amount of downtime required to perform the reconfiguration must also be considered.

Since most computational circuits are made up of sequential logic, a reconfiguration technique must be able resume operation in the last stable state before the fault. This means the dynamically reprogrammed FPGA must save state information, complete the repair through reconfiguration and reload state information before resuming service.

2.4 Current Research in FPGA Reconfiguration

There are two different styles of partial reconfiguration in current implementations and research [Xil04c]. The first is a module-based reconfiguration in which distinct portions of the FPGA are reconfigured while the remainder of the FPGA is active. Depending on the communication between modules, special consideration needs to be given to ensure proper I/O functionality between modules after reconfiguration. The second type of reconfiguration is difference-based partial reconfiguration in which custom bitstreams are used to change small sections of the device. Difference-based partial reconfiguration is useful for changing the contents of a LUT or switching to a different I/O standard during execution [Xil04c]. Each of these styles can be used for a number of applications and Virtex FPGA reprogramming has evolved to include both external reprogramming circuits and internal reprogramming circuits which take advantage of the ICAP. External reconfiguration circuits can use state of the art computer-aided design tools to produce partial bitstreams, while on demand production of partial bitstreams is one of the most difficult challenges in a dynamic reconfiguration system. Using externally produced partial bitstreams has been demonstrated as well as the use of pre-generated partial bitstreams but both required that the new FPGA configuration be known in advance [Xil04c].

FPGA reconfiguration systems can be categorized according to which device controls reconfiguration, the level of reconfiguration granularity, and when the configuration bitstream is generated [WB04].

2.4.1 Methods for Partial Reconfiguration. Partial reconfiguration can be controlled externally using the JTAG port of the FPGA. An external circuit, possibly a computer or another FPGA initiates the reconfiguration and loads the partial bitstream into the FPGA to reprogram it. This technique is used in embedded FPGAs that serve as computer coprocessors and reconfigured using the PCI bus.

Some FPGAs can initiate and internally control their own reconfiguration. A self reconfiguring platform implemented on both the Virtex-II and the Virtex-II Pro takes advantage of the platforms' embedded microprocessors to perform partial reconfiguration without external circuitry. It uses the ICAP, control logic, a small configuration cache and the MicroBlaze^{TM} embedded processor to support relocatable partial bitstreams. Relocatable partial bitstreams can be modified at run time to be implemented at multiple locations within an FPGA [BJRK$^+$03].

Hybrid configurations reprogram themselves, but the reconfiguration may be initiated internally or externally and the configuration bitstream may be retrieved from an external source, such as a bitstream server [WB04].

2.4.1.1 Module Based Dynamic Partial Reconfiguration.

Modular based partial reconfiguration methods for FPGAs have been used to develop Dynamically Reconfigurable Systems (DRSs) which actively reconfigure hardware based on previously generated bitstreams. By using these bitstreams to reprogram portions of the FPGA, unneeded parts of a system can be removed and replaced by another part. Figure 2.6 shows the layout of two reconfigurable modules in a Virtex-II Pro. Swappable modules are referred to as dynamic hardware plug-ins. A number of different configurations have been proposed for modular based reconfiguration platforms including both externally controlled and internally controlled systems [CCMM04].

For internal self-reconfiguration, the configuration controller must be implemented within the FPGA. Custom controllers have been implemented in the fabric of the FPGA for decoding secure bitstreams, for example a prototype using Blowfish encryption used 64% of the slices of a VC2V1000 device, but without Blowfish support only used 5% of the slices [FHA03]. Clearly the requirements for the controller (i.e., speed, security, connectivity) as well as the target FPGA, affects the size controller. Although custom controllers can be specialized for the target application, a more flexible approach uses an embedded processor on the FPGA. In an FPGA with hard microprocessors, the reconfiguration controller can use the microprocessors to maximize the reconfigurable portion of the FPGA available to the user's circuit.

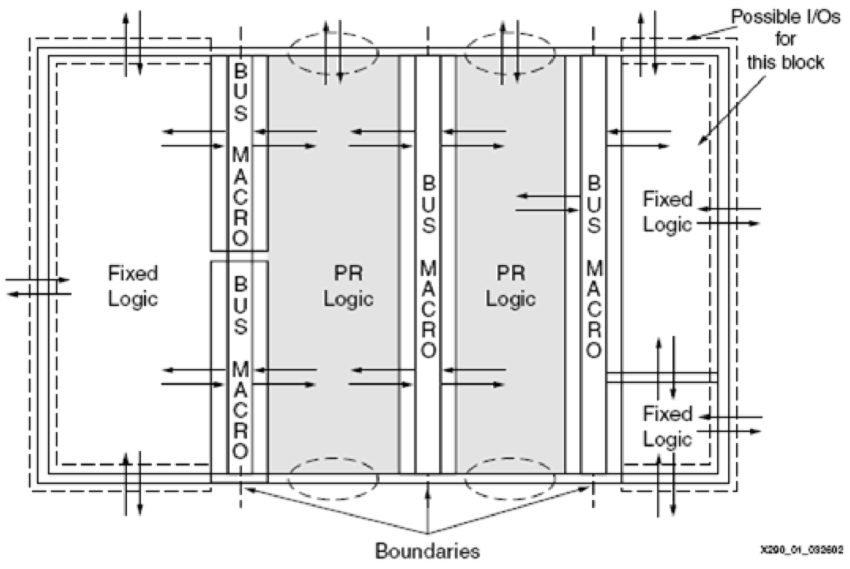


Figure 2.6: Design Layout with Two Reconfigurable Modules [Xil04c].

One dynamic reconfiguration system uses a MicroBlaze^{TM} embedded processor on the Virtex-II Pro [WB04]. Using uClinux, a version of Linux designed for microprocessors, bitstreams are retrieved from a remote server and used to reprogram the FPGA. A Linux driver provides an interface between applications running on the MicroBlaze^{TM} and the HWICAP core through the CoreConnect OPB on the Virtex-II Pro which allows scripts to be run within uClinux to perform partial reconfiguration using partial bitstreams from the server [WB04].

2.4.1.2 Difference Based Reconfiguration. By making small changes to the configuration bitstream, the behavior of the user circuit can be changed. Among other changes that can be made, the contents of a LUT can be altered changing the Boolean function performed by the logic cell. The transmit and receive characteristics of the RocketIO^{TM} Multi-Gigabit Transceivers (MGTs) on the Virtex-II Pro can also be changed in this manner [Xil04a]. By changing the parameters based on the run-time configuration, MGTs can compensate for unknown propagation delays at design time.

Table 2.1: Comparison of Core Generation Tools (adapted from [KJdlTR05]).

	Design Flow	JBits API based tools		Equations based tools		
Name	Modular Design	JPG	XPART	Core Unifier	PARBIT	BITPOS
Device	All Xilinx FPGAs	Virtex Series	Virtex II-Pro	Virtex Series	Virtex Series	Virtex II
CLB Reallocation	NO	NO	NO	NO	YES	YES
BRAMs/MULs Reallocation	YES	NO	NO	not specified	NO	YES
Controller Location	N/A	External	Internal	External	External	External
References	[Xil04a]	[RS02]	[BJKM+03]	[MMPM+03]	[HL01]	[KJTR05]
Approx Date	2004	2002	2002	2002	2002-2004	2004

2.4.1.3 Bitstream Manipulation on Self Reconfiguration Platforms.

Modern FPGA architectures support dynamic modification of a design, but there is a noticeable lack of design methodologies using non-proprietary tools to produce bitstreams required to reprogram an FPGA [DFR+05]. A number of software applications for the PC and for embedded processors have been developed and implemented on specific devices. The bitstream manipulation tools can be separated into two groups according to how they access the bitstream and produce the partial reconfiguration bitstream [KJdlTR05]. The first group uses application programming interfaces (APIs) to access previously generated bitstreams and manipulates them to produce the desired partial bitstream. The tools in the other group directly manipulate existing bitstreams to produce the desired bitstream. These tools are summarized in Table 2.1.

JBits, developed by Xilinx, provides an API to access the bitstream of select Xilinx FPGAs. JBits is a set of Java classes that provide an interface to operate on bitstreams generated by Xilinx design tools, or on bitstreams retrieved from programmed FPGAs. The original motivation of JBits was to support dynamic reconfiguration under software control. The API allows all configurable resources in the device to be programmed, thus providing direct support for dynamic reconfiguration [GLS99].

All action in JBits must be specified in the source code including routing. To make routing as fast as possible, JBits does not use the heuristics to solve the known NP-complete routing problems which produce routes that may not resemble the original circuit. Instead, JBits uses a library with access to all of the configurable architecture features of a device including CLBs, BRAM and all routing resources. These precompiled Java classes, specific to each type of device, produce the partial reconfiguration bitstreams [GLS99]. The last official version of JBits, JBits 3.0, extended support to the Virtex-II. However, JBits does not support the Virtex-II Pro.

A development that promises to expand the JBits support to other device families is the Alternative Wire Database (ADB), a supplemental connectivity database that interfaces with JBits to provide wiring information, routing and unrouting services. ADB extends JBits support to new FPGA families, including the Virtex-II Pro, and provides a router based on JHDLBits, an open source project that connects JHDL and JBits. JBits 3.0 does not include JRoute, a router than had been available in previous versions. ADB can generate configuration bitstreams when interfaced with JBits or a custom interface [SA04] and may be released with the next version of JBits.

For self-reconfiguring circuits, bitstream manipulation similar to JBits must be available on the embedded microprocessor. In Blodgett's self reconfiguring FPGA design, the software system relied on two APIs, the ICAP API and the Xilinx Partial Reconfiguration Toolkit (XPART). Since JBits is implemented in Java, it requires significant resources to run the Java Virtual Machine. Blodget's design using XPART is lightweight because XPART provides a minimal set of JBits API features implemented in C [BJRK$^+$03]. XPART has methods to read and modify select FPGA resources using ICAP and also provides basic support for the relocation of partial bitstreams. Unfortunately, XPART has never been released to the developer/research community [US05].

A number of additional tools have been developed to access and manipulate supported FPGA bitstreams using the JBits API. These tools include JBitsDiff, JBitsCopy, JPG, CoreUnifier and JHDLBits [KJdlTR05]. JBitsDiff generates cores from a full bitstream which can the inserted into another bitstream. JBitCopy extracts a core from a full bitstream file and merges it into another full bitstream. JPG uses files generated by Xilinx tools during design flow to extract cores. The JPG tool selects multiple partial bitstreams (cores) using a custom GUI and loads the FPGA through the JBits API [RS02]. Although these tools allow reconfiguration of CLBs, they do not *reallocate* cores. Reallocation allows a core to be placed at any location on the FPGA. Relocatable cores is a highly desirable capability in dynamic reprogramming of FPGAs since one partial bitstream can be used to generate multiple configurations instead of pre-compiling and storing partial bitstreams.

The second group of core generation tools directly manipulate existing bitstreams to produce partial bitstreams [KJdlTR05]. Using equations specific to the target FPGA, the location of an FPGA resource within the bitstream can be determined. Using an original bitstream, a target bitstream and parameters, including the coordinates for the source and destination of the core PARtial BItfile Transformer (PARBIT) can relocate a core by transforming and restructuring the partial bitstream [HL01]. BITstream POSitioner (BITPOS) provides a similar capability for the Virtex II family and includes the ability to reallocate BlockRAM memory space and embedded multiplier data [KJdlTR05]. A tool called Core Unifier, with JBits and equation base versions, has methods for inserting and connecting dynamic cores based on a common route wiring configuration [MMP$^+$03]. With the exception of XPART, all of these bitstream manipulation applications are external to the FPGA.

2.4.2 Hardware Bitstream Relocation. REPLICA2Pro is an implementation that relocates bitstreams internally. REPLICA2Pro relocates partial bitstreams in

Virtex-II (Pro) devices by changing the addressing of the configuration frames in hardware. The hardware decodes the partial bitstream finding the frame addresses and adjusting them to implement the module in a new location. To avoid extra configuration time, the REPLICA2Pro filter is inserted between the configuration manager, which selects the bitstream and the offset distance, and the ICAP interface. REPLICA2Pro relies heavily upon custom software to prepare the FPGA creating the infrastructure to support reconfigurable modules. These programs generate the communication infrastructure and clock trees to ensure the module will be properly connected in its new location [KP06].

2.4.3 Automatic Dynamic Active Partial Reconfiguration for Fault Tolerance.

A number of methodologies have been developed to enhance the yield of FPGAs during the fabrication process using fault tolerance methods. A good summary of these techniques can be found in [Ive06]. While based on similar techniques, active dynamic reconfiguration operates at runtime.

To use SRAM-based FPGA for dynamic fault tolerance, the system must meet the following objectives [XSHL99]:

- Overhead must be as low as possible. The FPGA area used by the reconfiguration circuit and during reconfiguration must be minimized since it determines the maximum size of the user circuit and how many times it can be reconfigured.
- The replacement algorithm must be simple and have the shortest execution time possible.
- The circuit after fault recovery must still meet functional and performance requirements.

A number of reconfiguration schemes which reduce overhead associated with reconfiguration have been proposed and demonstrated. One reduces the complexity of reconfiguring by partitioning the physical design into tiles [LMSP99]. Multiple configurations are generated for each tile—all of which perform the same function and have the same connectivity with neighboring tiles. Reliability is achieved by having multiple configurations of each tile that do not use certain resources within the tile. The tile configurations are generated at design time and stored in memory for access at run-time.

Assuming the location of the fault is known, a configuration that avoids the faulty resource is retrieved from memory and used to reconfigure the device. Spare interconnects between tiles can be reserved and activated if an interconnect fault is identified. Compared to redundancy-based fault-tolerant techniques, the overhead for this approach is low since redundant modules are not implemented in the FPGA array. Although the regular structure of FPGAs makes them good platforms for this approach, the implementation is architecture specific.

For the Xilinx architecture, each tile is made up of groups of CLBs. The number of unused CLBs and interconnects in the tile determines the amount of redundancy and overhead. Multiple configurations for each tile are generated at design-time– each leaving a portion of the tile unused (i.e., CLBs and interconnections). Although overhead is directly related to the size of the tiles and number of spare resources, the area and reconfiguration time for this method is low. Tests of 9 circuits on a Xilinx FPGA indicate timing and area increased between 17% and 45% and between 2.6% and 10.2% respectively [LMSP99]. Although this approach reduced system downtime since alternate configurations are readily available in memory, the memory required to store the alternate configuration is many times greater than the original configuration size.

A robust configurable system design with build-in self-healing (BISH) highlights many of the considerations that must be made for a SRAM-based fault-tolerant system [GAF05]. Since many of the observations made in the design have implications for SRAM-based fault-tolerant systems, they are summarized below.

As noted in Section 2.1, today's FPGAs are susceptible to two types of errors: soft errors, or SEUs, which are transient errors caused by radiation; and hard errors–

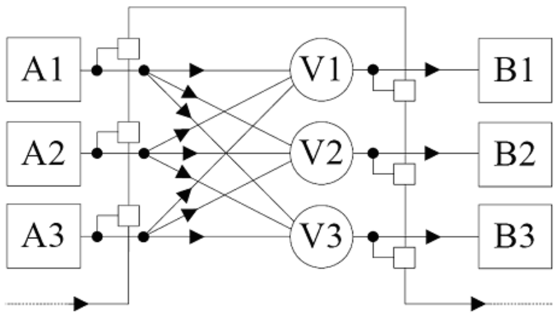


Figure 2.7: TMR with boundary scan [GAF05].

the result of permanent physical damage to the FPGA. Since SRAM memory is used for both configuration memory and data memory in a SRAM FPGA, soft errors can change both the function of the FPGA and the data stored within it.

Unlike previous fault tolerance approaches, the approach below includes detection, diagnosis and repair. To prevent faults from propagating through the system TMR masks faults and reconfiguration replaces modules that have suffered a hard error, similar to the N-modular redundancy with K-sparing approach discussed in Section 2.2.1. In a traditional TMR circuit, it is difficult to determine which module is faulty since the TMR circuit masks the fault. To determine the faulty module is faulty, a boundary scan configuration can be added to the TMR circuit as shown in Figure 2.7.

The boundary scan allows a microprocessor to analyze the output of each module (A1-A3) and the output of each of the voters (V1-V3). If one of the redundant modules has a different output, a fault is presumed to be causing the error. If all module outputs are equal but the voter outputs are not equal, the fault is presumed to be in the voters. Thus, the appropriate actions to take and how to repair of the circuit can be determined.

Assuming no errors in retrieving and analyzing the output of the modules and the voters with the boundary scan (a methodology for validating this is explained in [GAF05]), the next step is to determine if the error is a soft or hard error. If it is a soft error, it should automatically correct itself next time the registers within the module is updated. Since voters are normally implemented in combinational logic, this type of error will not affect them. If the error is not resolved after the registers have updated, there are two possible causes for the error. A soft error in the configuration memory has caused the behavior of the module or voter to change or a hard error within the module or voter.

If there is a soft error in configuration memory, it can be detected by extracting a partial bitstream from configuration memory and comparing it with the original bitstream or by checking the bitstreams CRC. If an error is detected, a partial bitstream can be reloaded to configuration memory, repairing the configuration memory. Once reconfiguration is complete, boundary scan can determine if reconfiguration was successful.

If an error in the configuration memory is not detected, the most likely cause of the error is a physical defect in the array. Physical defects in the array can not be repaired and reconfiguration must remap the module to a fault free area of the FPGA. Although the TMR configuration has masked the module error from the rest of the FPGA, remapping the module restores the reliability index of the circuit.

To maximize resources, once a portion of the FPGA has been released by remapping the module it contained, the embedded microprocessor can diagnose the released resources to determine exactly which resource is faulty. By keeping track of precisely which resources are defective, the microprocessor can maximize the use of the FPGA by allowing modules that do not use an affected resource to be mapped to that area.

With TMR masking faults, there is some flexibility in the timing of detection, diagnosis and repair actions. Although there is overhead associated with each of these operations, the proposed BISH system performs these operations as background tasks on the microprocessor, minimizing the circuitry dedicated to BISH.

Since the microprocessor is vulnerable to soft and hard errors, it is also implemented using TMR. Each microprocessor is broken up into small modules, and a malfunctioning microprocessor relies on the other two microprocessors to replicate the malfunctioning module, remove it from service and replace it.

As with other proposed approaches, this approach has not been implemented and relies upon JBit-based tools for reconfiguration which are under development.

Although this approach provides effective solutions for recovery from a number of different soft and hard errors, its developers acknowledge a number of vulnerabilities such as errors in the configurations control circuit, the ICAP and the boundary scan architecture [GAF05]. Protections other than reconfiguration must ensure that the reconfiguration system is available when needed.

An alternative to pre-compiled tiles or dynamically generated configuration bitstreams is reconfiguring the FPGA using bitstreams based on precompiled columns [HM01]. This technique has a fast reconfiguration time since routing is not determined dynamically. The regular structure of Xilinx FPGAs, mean they have the same circuitry, routing resources, and configuration architecture in every CLB column which results in highly correlated bitstreams. Thus, multiple bitstreams can be compressed. Two schemes for column-based reconfiguration and bitstream compression are proposed in [HM01].

The overlapping scheme relies on a base configuration being mapped into column-based functional modules. The function of the circuit is defined by the modules and their interconnections. As shown in Figure 2.8, unused columns in the base configuration leave room for alternative configurations which remap the modules.

Since the structure of each CLB column is the same, groups of column-based modules can be shifted and the only additional reconfiguration needed is to repair the interconnections between groups. To reconfigure from the base configuration to alternative configuration in Figure 2.8b, functions C and D are shifted to columns 4 and 5 and the interconnections between function B and function C were restored. Since functions C and D remain in adjacent columns, interconnections between the two column-based modules are intact.

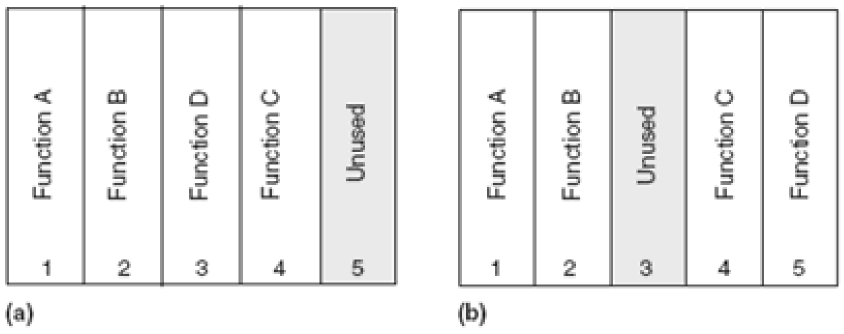


Figure 2.8: The Overlapping Precompiled Column Scheme. (a) Base configuration with column 5 intentionally unused (b) Alternative configuration with column 3 intentionally unused [?].

The number of unused CLB columns determines the fault tolerance of the FPGA. To tolerate m faults, m spare columns are required. If the base circuit configuration required k columns to implement the user function, the overlapping scheme required $m+k$ CLB columns to map an m-tolerant configuration. To achieve a m-column tolerant design, $C(k+m, m) = (k+m)!/(m!k!)$ configurations (including the base configuration) must be available or be calculated at runtime [HM01]. However, since alternate configurations are generated by shifting the column-based modules, the bitstreams are similar and can be compressed and memory overhead reduced. Details on the compression technique can be found in [HM01].

If the user circuit can be implemented in less than 1/2 of the FPGA's columns, an alternative approach maps the entire user circuit into unused portions of the FPGA during reconfiguration. In this scheme, the entire circuit is shifted into an unused portion. For a circuit to be m-column tolerant it must be mapped in $1/(m+1)$ or less of the entire FPGA columns. This approach uses less memory than the overlapping scheme because there are only $m+1$ configurations (including the base) and since the entire circuit is shifted as a block, the relative position among the column-based modules is preserved in all configurations.

For both column-based module schemes, circuit performance, in terms of worst case critical path, increases from 11% to 18%. For 4 test circuits, the minimum storage overhead ranged from 15% to 35% for the over-lapping scheme and 2% to 6% for the non-overlapping scheme [HM01].

Using a column-based approach the degree of fault tolerance is constrained by the number of columns in the FPGA and in the user's circuit, as well as how the user's circuit can be divided into column-based modules. Because a frame is the smallest reconfigurable segment, if reconfiguration is performed at the frame level, the degree of fault tolerance can be greatly improve over the column-based approach.

One advantage of a column-based approach is the location of faults. Unlike fine-grained approaches which try to determine which CLB or frame has a fault, faults only need to be determined to be in a particular column. An alternative to fault location is to try possible configurations until a configuration that works properly is found [HM01]. Although this approach lengthens reconfiguration time, reducing availability, it eliminates the need for fault detection hardware.

Although single FPGAs can recover from faults in the user circuit using reconfiguration, they are vulnerable to errors in the logic implementing the reconfiguration circuit. The reconfiguration circuit can be made fault tolerant through traditional hardware redundancy or a dual-FPGA configuration can be used. Expanding on the column-based approach, a dual-FPGA reconfiguration architecture allows the system to recover from all types of soft errors.

In the dual-FPGA configuration, each FPGA runs user applications and uses soft microcontrollers so each FPGA can be reconfigured [MHS$^+$04]. The microcontroller on each FPGA reprograms the other FPGA. User applications mapped on the FPGA must include error detection and autonomous recovery techniques to maintain proper operation. Once a non-recoverable error is detected and reported to the microcontroller, the microcontroller reports the error to the microcontroller on the other FPGA and the second FPGA reconfigures the first FPGA.

Since temporary errors are more common than permanent faults, a soft error is assumed and the second FPGA validates then corrects the configuration bits of the first FPGA if necessary. If an error persists once execution of the first FPGA is resumed, a permanent fault is presumed and the second FPGA reconfigures the first FPGA using a modified column-based pre-compiled reconfiguration scheme to avoid the fault [MHS+04]. Since error detection is incorporated into the user circuit, the number of new configurations to be tried is reduced based on the location of the error detected.

The dual-FPGA approach also allows for an alternative to TMR which adds three microcontrollers to each FPGA (using considerable area). Instead, TMR concurrent error detection (CED) signals designed into the microcontrollers can determine if the other FPGA's microcontroller has an error and requires reconfiguration [MHS+04]. This approach can be expanded to include the entire reconfiguration circuit making the dual-FPGA architecture capable of recovering for temporary or permanent errors to both the user circuit and the reconfiguration circuit.

There are a number of resources within an FPGA vital to its proper operation that can not be corrected through reconfiguration. These include external connections (I/O pins) and the actual reconfiguration circuitry on the FPGA including the JTAG, serial, or SelectMap interfaces. Reconfiguration can be used to avoid some of these resources. In a single FPGA architecture, a fault in the ICAP would prevent the FPGA from reconfiguring. However, in a dual FPGA architecture there are multiple interfaces and configurations which use an alternative reconfiguration mode (JTAG or serial) can be designed and implemented if an error in the FPGA reconfiguration interface is detected or suspected [MHS+04]. Additionally, if the design permits, configurations with alternate I/O can be compiled.

2.5 Summary

Reconfiguration of FPGAs and active fault tolerance techniques improve reliability. The capability, quick development time and relatively low cost of today's FPGAs make them an attractive platform for computing applications, but without effective, low-overhead methods for making them more dependable, their applications in high radiation environments are limited.

Multiple approaches for fault tolerance through reconfiguration have been proposed and some have been demonstrated. Since each approach is target specific, and the overhead and improvement in reliability varies by benchmark and size of the target FPGA used, direct comparison is difficult. Due to the amount of time needed to generate bitstreams dynamically, many of the techniques use pre-compiled partial bitstream or portions of the bitstream to be reassembled at runtime.

Although there are a number of techniques to generate partial bitstreams external to the FPGA being reprogrammed, no applications are readily available to dynamically generate partial bitstreams within the FPGA. If the resources to be avoided are known, dynamically generating partial bitstreams to reprogram the FPGA with frame granularity, can increase the number of recoverable faults since the number of spares would be maximized. Even so, the techniques developed to date focus on efficient fault recovery and avoid dynamic generation of bitstreams due to the considerable time involve and large memory overhead of available tools.

III. Development of a Dynamic Reconfiguration System

3.1 Introduction

Using partial reconfiguration as a method for fault tolerance adds adaptability and flexibility to the system but also increases complexity. A fault tolerant system must not only be able recover from a fault, it must be able to detect the fault. Although detecting and repairing faults with frame granularity maximizes the number of recoverable faults, detecting the fault and generating a partial bitstream to repair the fault would be very difficult. Column-based modular reconfiguration performs fault detection and recovery at the module level. Faulty modules are replaced by modules in use at another locations or altered to avoid the faulty resource.

Most previous reconfiguration based fault tolerant systems store separate bitstreams for each area on the FPGA that the module can be placed, even if the modules are functionally equivalent. These static bitstreams are pre-generated and stored in memory or perhaps externally until needed. Since the bitstreams are for specific locations on the FPGA, separate bitstreams for each location on the FPGA a bitstream that targets that location on the FPGA must be available. This requires a large amount of memory which typically is not available in microprocessing platforms.

3.2 Problem Definition

3.2.1 Goals and Hypothesis. The primary goal of this research is to develop an efficient fault recovery system that allows a user circuit to operate through faults. The system will use relocatable modules to recover from faults without storing individual bitstreams. Given the location of a fault in one of the relocatable modules, the system will automatically replace the faulty module by properly translating the bitstreams for the module and programming the FPGA through the ICAP. A user circuit is considered to operate through a fault if it continues to operate properly despite a fault occurring. A secondary goal is to evaluate column-based reconfiguration techniques which take advantage of relocatable modules.

This research will determine whether the architecture of the Xilinx Virtex-II Pro, and the Xilinx partial reconfiguration toolchain, can implement such a system and whether reconfiguration methods can be developed to take advantage of the architecture of the Virtex-II Pro.

3.2.2 Approach. To eliminate the need to store multiple partial bitstreams for each module, that is, a separate bitstream for each possible location the module could be placed within the FPGA, a method is developed to relocate the core by manipulating the partial bitstream with an embedded microprocessor. Using this approach, only one partial bitstream for each module needs to be stored in memory, minimizing memory usage. To minimize the FPGA area dedicated to relocation, all calculations needed to manipulate the bitstream are performed using a embedded microprocessor. In an operational system the microprocessor could be used for other tasks when not needed for reprogramming.

To achieve user circuit operation through faults, the user circuit is implemented using TMR. Assuming the location of a fault is known, the microprocessor generates a partial bit stream by manipulating an existing partial bitstream stored in memory for the module determined to be faulty and relocating and reconnecting the replacement module. Three TMR configurations that take advantage of relocatable modules are developed.

3.3 A Column-Based Fault Tolerant Configuration

A column-based modular approach can be used to implement a fault-tolerant circuit that operates through faults. Partial bitstreams implement replacement modules in spare locations to repair the circuit when a fault is discovered. The reconfigurable modules that perform the primary function of the circuit are referred to as functional modules. Figure 3.1 is a basic TMR circuit with three active functional modules and room for two spare modules. Since the functional modules connect directly to the re-

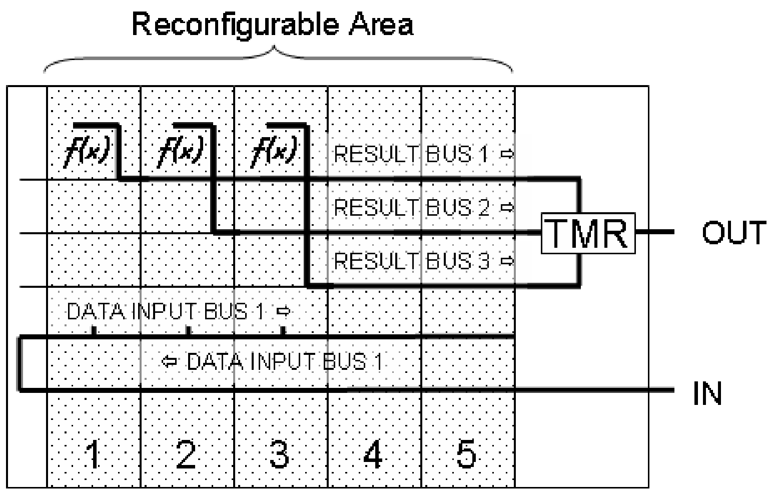


Figure 3.1: Basic TMR Design. This TMR circuit contains three modules (1, 2, and 3) perform the same function, denoted *f(x)*. Their results are send to a voting circuit which determines the consensus output. Modules 4 and 5 are spares which pass through the results. The input to each of the modules is delivered through a data input bus.

sult busses, this configuration is referred to as the "direct connect" design throughout the remainder of this document

3.3.1 Benefits. Using TMR provides two key benefits. First, TMR provides passive fault tolerance masking the fault and preventing errors from propagating into other parts of the system. Assuming only one module is faulty at a time and the two other modules continue to run correctly, the TMR circuit will select the output of the two correctly functioning circuits. Although translating the bitstream and reprogramming the FPGA takes time, the masking ability of the TMR circuit allows the circuit to continue to produce the correct result. The second benefit is the detection of errors. If two of the three modules are producing the same result, the module that does not match the other two must have an error and should be replaced. Although it is assumed that the location of the fault is know, a method similar to the boundary scan techniques in the BISH design [GAF05] could be used to determine the location of the fault. Once the module is replaced redundancy is restored and system is ready for another fault.

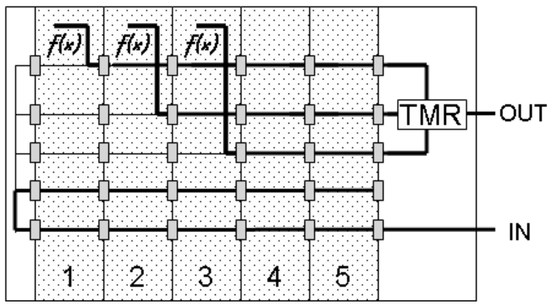
3.3.2 Routing and Timing. Reconfiguring a circuit introduces two related problems, routing and timing. Once a module has been relocated it must be reconnected to the TMR circuit. All routing in the FPGA design is typically performed by implementation tools prior to programming the FPGA. Although dynamic rerouting has been demonstrated using JBits and ADB [SA04], all signals entering and exiting a reconfigurable areas must pass through bus macros (cf., Section 3.8.2). Thus an alternative solution for modular reconfiguration is needed.

One solution is to have multiple partial bitstreams which perform the same function but are connected to different busses. Each data bus used in the TMR design is labeled in Figure 3.1. Modules 1, 2 and 3 all perform the same function, represented by *f(x)*, but are connected to different busses. The busses carry the result from each module to the TMR circuit where their results are compared. This configuration eliminates the need for rerouting the design after reconfiguration.

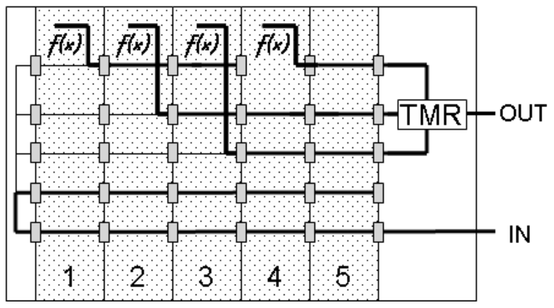
When using a TMR circuit, the results from each source must arrive within the same clock cycle. The three result busses carry the results from the functional modules to the TMR circuit. The data input bus in this configuration provides the same combined path length for the input and results signals no matter which location the function module is placed in. This ensures that timing is not affected by the location of the module. The input signal passes through each of the reconfigurable modules then loops back to the static module. The function implemented by the module receives input from the input bus as it passes through the modules the second time.

3.4 Using Relocatable Modules in TMR Designs

To implement a functional modules on an FPGA only one reference bitstream is needed. This bitstream can be altered to allow the module to be placed at any location on the FPGA. Using a technique similar to how bitstreams are manipulated in the REPLICA2Pro [KP06], column-based modules can be relocated using software to move them to any location on the FPGA.



(a)



(b)

Figure 3.2: Modular Functions Before and After Reconfiguration. Bus macros are shown between modules. (a) Three modules 1, 2, and 3 have identical functionality but are connected to three different busses feeding the TMR circuit. Modules 4 and 5 are spares but pass data. (b) The bitstream used to program location 1 has been translated used to program location 4 changing which module produces the result on bus 1 that reaches the TMR circuit.

Relocatable modules greatly reduce the memory needed to store bitstreams. The module is relocated by altering its bitstream to change the target location. This technique reduces the number of bitstreams needed to implement a module in n locations from n to 1. A method for relocating bitstreams is developed in Section 3.6.

3.4.1 Bitstream Storage Savings With Relocatable Modules. Comparing Figures 3.2a and 3.2b it can be seen that by moving module 1 into the location of module 4 not only is the functionality of module 1 replicated but it is also properly connected to the bus. This also prevents the faulty results of the module at location 1 from reaching the TMR circuit. Although storing and relocating multiple version of each functional module is a convenient way to reconnect modules in a dynamic partial reconfiguration system, the ability to place a module at multiple locations and connect to multiple busses increases the number of bitstreams needed. Without bitstream relocation, the number of bitstreams needed is

$$\textit{\# of bitstreams} = \textit{\# of functions} \times \textit{\# of locations} \times \textit{\# of busses}. \tag{3.1}$$

Although this technique allows a functional module to be placed in any location on the FPGA using only one bitstream, separate bitstreams to connect the module to different busses are still needed. With detailed knowledge of how the bitstream establishes connections between CLB blocks it is possible to establish new connections within the FPGA by manipulating the bitstream bit-by-bit. However, since the information required about how routes are connected in the FPGA is not readily available, techniques described in Sections 3.4.2 and 3.4.3 have been developed for dynamic routing which eliminate the needed for intricate knowledge of the FPGA and the need for separate versions of the functional module for each bus connection. The first is based on column-based partial reconfiguration and the second uses difference based partial reconfiguration.

3.4.2 Routing with Relocatable Interconnect Modules. To support dynamic routing, interconnect modules can be added to the partial reconfiguration area as in Figure 3.3. By adding separate modules to perform bus routing, each functional module has a standard configuration. The output of each functional module is passed through the bus macro in the upper right corner of each functional module and the data on each of the busses passes through. The interconnect modules take the output of the functional modules and connect it to the appropriate bus while allowing the data on the other busses to pass through unchanged. Note that this configuration can easily be expanded by adding additional busses.

Similar to the partial bitstream used to instantiate the functional modules, the bitstreams for reference interconnect modules can be altered to change where the module will be placed. Using interconnect modules, reconfiguring the circuit consists of relocating the functional module followed by relocating the interconnect module that connects it to the proper result bus.

The benefit of using module relocation depends on the number of functional modules, the number of possible locations for the functional modules and the number of busses they connect to. The number of column-based bitstreams needed using

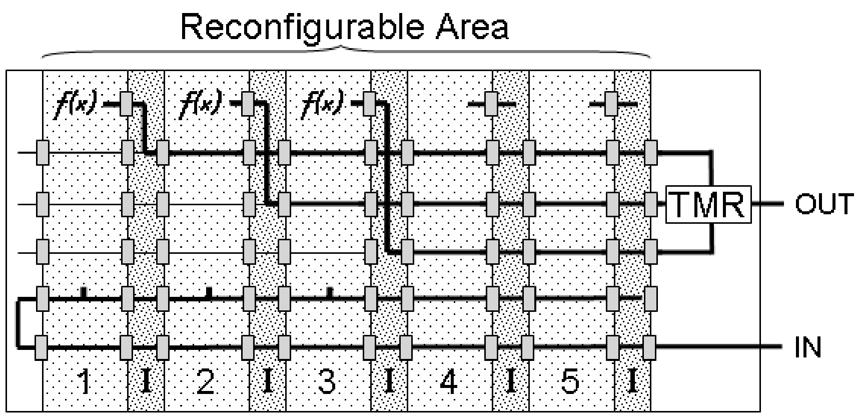


Figure 3.3: Relocatable Functional and Interconnect Module Configuration. Using interconnect modules, labelled I, allows the all functional modules to have a standard configuration by eliminating the need for different versions that connect to the different result busses.

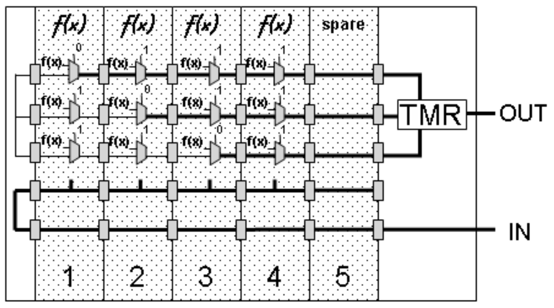
interconnect modules for dynamic routing is

$$\# \textit{ of bitstreams} = \# \textit{ of functions} + \# \textit{ of busses.} \tag{3.2}$$

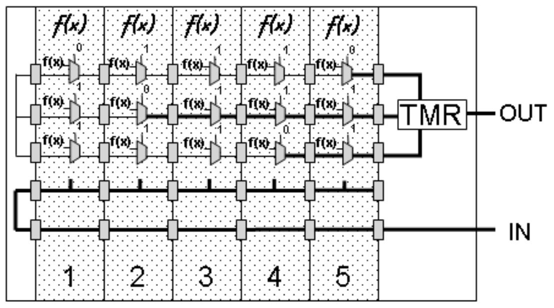
Only one partial bitstream is needed for each functional module and each interconnect module since they can be relocated to the desired location.

In (3.1) all of the bitstreams are approximately the same size. Note that the bitstreams for interconnect modules in (3.2) may be much smaller than bitstreams for the functional modules. Limitations that reduce the benefit of interconnect modules are examined in Section 4.2.1.

3.4.3 Rerouting Using Difference Based Reconfiguration. The other form of partial reconfiguration is difference-based partial reconfiguration. When there are small changes between two designs, a partial bitstream can be produced that only reflects the changes between the two designs. Difference based partial reconfiguration creates a partial bitstream by comparing two bitstreams and determining which frames are different between them. The partial bitstream only reprograms the frames that have changed.



(a)



(b)

Figure 3.4: Relocatable Modules with LUT Selected Bus Connections. In functional modules, bus connections are controlled by LUTs which provide inputs to a multiplexer to select between the module's function and the incoming results bus. The output of the LUTs shown as zeros or one above. (a) Modules 1, 2, and 3 provide the inputs to the TMR circuit. Module 4 illustrates having a preprogrammed module that does is not connect to the bus system. Module 5 is a spare module that passes bus signals unaltered. (b) Modules 2, 4 and 5 now provide the inputs to the TMR circuit. The LUTs in module 4 have been changed to connect to bus 2 and module 5 has been reprogrammed using the relocated partial bitstream.

Like the interconnect module approach, this technique changes which bus each functional module connects to. Consider two versions of a functional module. The first connects to results bus 1 and another that connects to results bus 2. Since the only functional difference between the two is which bus they connect there may be little difference between the two partial bitstreams. If this is so, a difference-based partial bitstream would be small. However, since each of the modules are placed and routed independently and optimized based on the location of their output, the difference between the two could be dramatic requiring a larger partial bitstream to account for all of the changes.

To prevent large differences between the bitstreams for functional modules that connect to different busses, a multiplexer shown in Figure 3.4a and 3.4b selects which bus each functional module places its output on. The bus selected by the multiplexer is determined by the value of LUTs. To change which bus the module is connected to, the values in the LUTs are changed using partial reconfiguration.

To ensure that the only difference between functional modules that connect to different busses is the change in the LUTs, the modules that connect to different bitstreams are created by editing the Native Circuit Description (NCD) file for the functional module in location 1. The NCD file contains a physical representation of the design mapped to specific resources in the target FPGA. The modified NCD file of the functional module in location 1 that connects to results bus 1 results in a functional module that passes through all signals. From these two NCD files, a difference-based partial bitstream is generated which changed the values in the LUTs.

Unlike the previous two approaches, LUT-based routing requires special care to prevent the relocated module from connecting to the wrong results bus. To prevent result bus contamination, the initial configuration of the relocatable functional modular must pass signals on the result busses unaltered. Once the functional module has been placed, the partial bitstream to change the values in the LUTs, selecting the proper results bus, can be relocated and applied.

3.5 The Target FPGA

The Xilinx University Program Virtex-II Pro (XUPV2P) development system is used to test the relocation technique and fault-tolerant modular configurations. Board revision C uses a XC2VP30 Virtex-II Pro. For extended memory, 256MB of PC2100 Infineon DDR SDRAM (part number: HYS64D32000GU-7-B) is used and a 64MB DG Vision Compact Flash stores the Advanced Configuration Environment (ACE) file to program the FPGA and store reference partial bitstreams during testing.

In addition to the complexities of fault tolerance, the target XUPV2P board introduces constraints that must be taken into consideration in when making a module relocatable. The architecture of the Virtex-II (Pro) restricts the size and shape of the reconfigurable modules. Although reconfigurable area can be defined as a small rectangular area on the FPGA such as the one labeled “A” in Figure 3.5, since the programmable frames span the length of a column all affected frames must be reprogrammed. For example, to reconfigure module “A”, a partial bitstream that

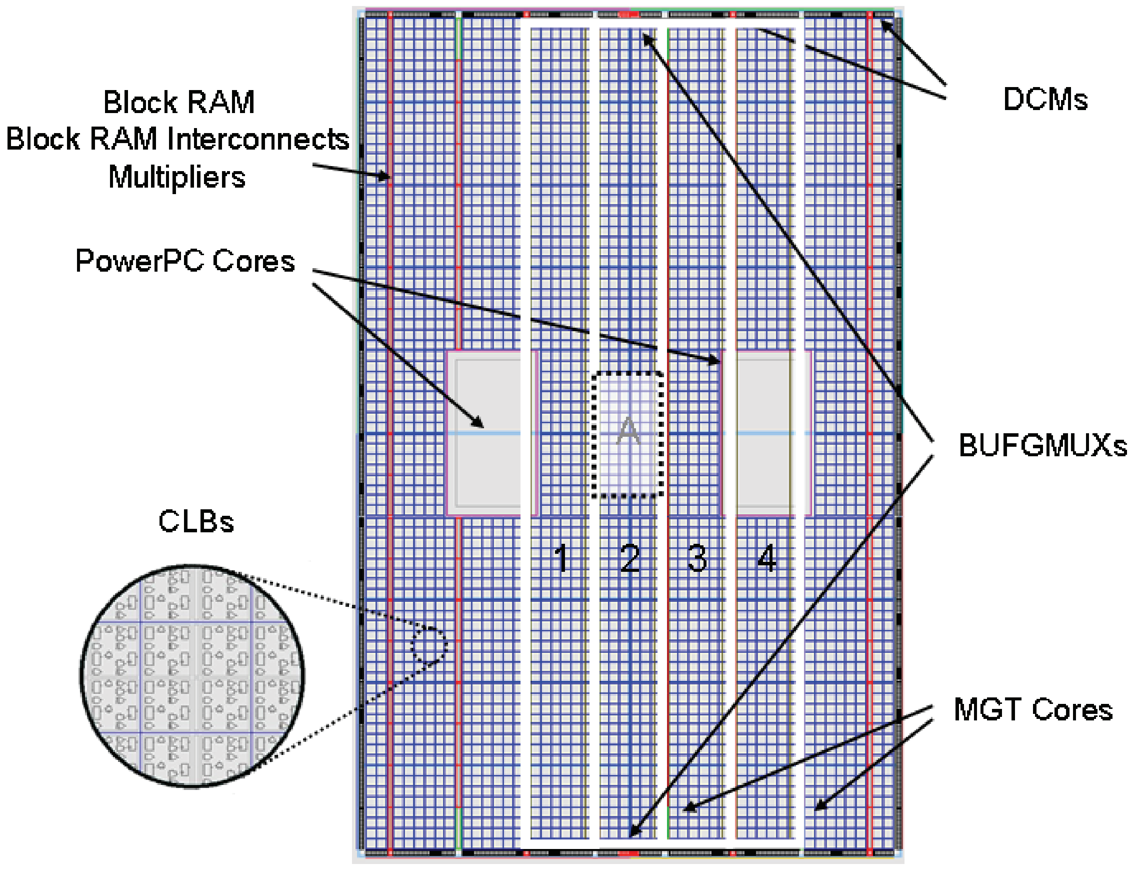


Figure 3.5: Layout of Virtex-II Pro (xc2vp30ff896-7) in PlanAhead. The majority of the FPGA is made up of CLB blocks. Four notional modules are labelled 1-4. The inclusion of PowerPC cores and MGT make the CLB and BRAM resources available to column-based modules vary by location. Additionally, BUFGMUX and DCM resources are limited. Pin connections are made through the I/O banks bordering the FPGA. Note that not all instances of each type of resource are labelled. The resources are placed symmetrically horizontally and vertically.

contains all of the configuration data for module 2 with the changes to module "A" is needed since each frame spans the entire column assuming the change in module "A" affects every frame in the column. Otherwise, only affected frames are included in the bitstream. The logic outside of module "A" can continue to operated during configuration because the Virtex-II Pro offers "glitchless" reconfiguration. That is, if a configuration bit holds the same value after reconfiguration as it did before, the resource it programs will not "glitch" [BJRK$^+$03]. Exceptions to this behavior in the Virtex-II (Pro) are the LUT RAM and SLR16 primitives [SBB$^+$06].

Since the bitstream to reconfigure a module contains all affected frames, special consideration must be made for the size and location of reconfigurable modules to ensure they can be relocated. Consider, for example, 4 equal size modules have been labeled 1-4 in Figure 3.5. Although the modules are the same size, the resources available at each location varies. For example, consider moving module 2 which includes submodule "A" to location 4. Due to the PowerPC, the CLB and BRAM resources needed by submodule "A" may not be available in location 4. Further inspection shows that, in this configuration, no two modules have the same resources available due to the PowerPCs. In addition to resource availability irregularities, there are a number of unique resources in the Virtex-II Pro. In addition to the resources labeled in Figure 3.5, Input/Output Blocks surround the perimeter of the FPGA.

The configuration bitstream does not contain bits to program any portion of the PowerPC but static inputs control adjacent BRAM columns. The XC2VP30 also has Multi Gigabit Transceivers (MGT) cores which replace a portion of the BRAM. These MGT cores are programmed using approximately 300 configuration bits in the adjacent BRAM interconnect column [Xil05b]. Although the MGT cores may not be used in a design, it should be recognized that relocatable modules can not rely on the BRAM components to be consistent between all locations.

Despite these shortcomings, the XUPV2P board makes an acceptable test platform because of excellent documentation and demonstrated use in previous partial reconfiguration experiments. Additionally, the XUPV2P can be reprogrammed through multiple interfaces including the ICAP, JTAG and SystemACE.

3.6 Developing the Bitstream Translation Program

Following a column based modular design approach [Xil04c,HM01], the location of the reconfigurable modules is a function of the frame addresses in the bitstream. This is also true for difference based bitstreams, but only frames that are different between the two bitstreams are reprogrammed. To relocate a module or a difference based bitstream, only the frame addresses and CRC values needed to be changed.

					BA		MJA								MNA								Byte Number								
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	0	0	0	0

Figure 3.6: Frame Address Composition. The type of column is determined by the BA, the columns is determined the MJA and the frame within the column is determined by the MNA [Xil05b].

Before a method is developed for translating the bitstream, the composition of the bitstreams and the configuration memory addressing scheme in the Virtex-II Pro is addresses must be understood.

3.6.1 Virtex-II Pro Bitstream Composition. The details about the composition and construction of the Virtex-II Pro and Virtex-II Pro X bitstreams are in [Xil05b]. The following analysis and decomposition of the bitstreams relies heavily on this document. To translate a bitstream, the format and structure of the entire bitstream must be understood so that the addresses can be located and altered appropriately. The bitstream translation program (BTP) assumes the original partial bitstream is valid and contains all commands necessary in a partial bitstreams to reprogram the FPGA.

Following an initial 32-bit synchronization sequence, the remainder of the bitstream is made up of multiple data packets. The format of the bitstream packets is discussed in Section 3.6.3, but first the addressing scheme is presented.

3.6.2 Configuration Memory Addressing. Each configuration frame is addressed using a unique 32-bit frame address. The 32-bit address is composed of the block address (BA), major address (MJA) and a minor address (MNA). The MJA specifies the column and the MNA specifies a specific frame within a column. Figure 3.6 shows how the BA, MJA, MNA and byte address make up the frame address. The relationship between the BA, MJA and MNA is illustrated in Figure 3.7 where n is number of devices CLB columns and m is number of device BRAM/BRAM Interconnect Columns. The values for n and m are device dependent.

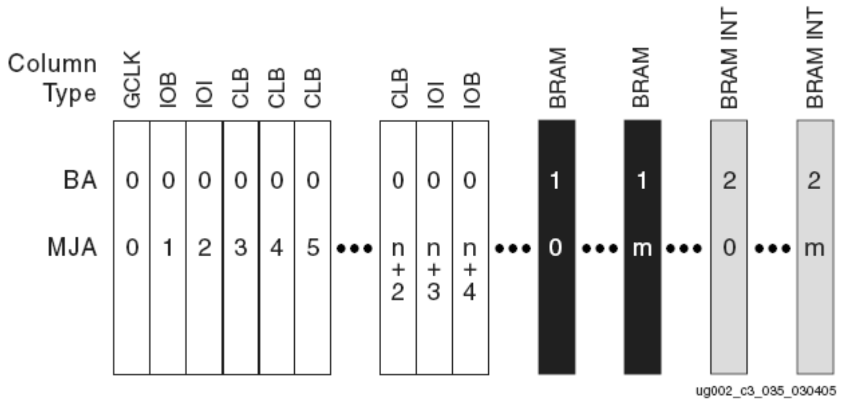


Figure 3.7: Column-Level (MJA) Configuration Memory Map [Xil05b].

Divided by BA, the configuration memory for the Virtex II-Pro is independently addressable in three blocks [Xil05b]:

- **Block Address 0** (BA 00) contains all Global Clock (GCLK), Input/Output Block (IOB), Input/Output Interface (IOI), and CLB configuration columns
- **Block Address 1** (BA 01) contains all BRAM columns
- **Block Address 2** (BA 02) contains all BRAM interconnect columns

All configuration memory is programmed through a bitstream. To change the addressing of the bitstream, the processing of the bitstream must be understood.

3.6.3 Bitstream Packet Type. Bitstream packets are used to write data to the registers of the FPGA configuration logic. Configuration packets can set configuration options, program configuration memory, or change the value of internal FPGA configuration signals. Each bitstream packet contains a header and a body. Two types of packets are used.

Type 1, is used for smaller packets, up to $2^{11} - 1$ words, and Type 2 is used for larger packets, up to $2^{17} - 1$ words. As can be seen from the packets headers in Figures 3.8 and 3.9, both Type 1 and Type 2 packet headers can be used for reading and writing, but only Type 1 packets can specify a register address. Type 2 packets

Type			WR	RD	Register Address														RSVD		Word Count										
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x

Figure 3.8: Bitstream Packet Type 1 [Xil05b].

Type			WR	RD	Word Count																										
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Figure 3.9: Bitstream Packet Type 2 [Xil05b].

must be used directly after a Type 1 packet and read or write to the register specified by the Type 1 packet.

3.6.4 Software Emulation of the Packet Processor. The packet processor is the portion of the FPGA configuration control logic that drives incoming data into the configuration register targeted by the packet header. The packet processor continues to drive incoming data to the targeted register until the word count set by the packet header reaches zero, signifying the end of the packet. The BTP must keep track of how many words have been processed to determine when one packet ends and the next one begins. Once the packet processor finishes processing a packet it waits for the arrival of the next packet. This process is repeated for each packet until the end of the bitstream is reached.

For Type 1 instructions, the word count is the 11 LSB of the packet header. For Type 2 instruction, the word count is the 27 LSB of the packet header. A Type 2 write instruction is always preceded by a Type 1 header indicating that zero words will be written to the Frame Data Input Register FDIR). Both Type 1 and Type 2 instructions can be used to read or write, but for the application of relocatable modules the partial bitstreams, all packet headers have write commands. The translation program as implemented in Appendix B supports changing the address bitstreams that both read from and write to memory.

Table 3.1: Writing to Configuration Memory with CRC [Xil05b].

Configuration Data	Explanation
30008001	Type 1 Packet Header: Write 1 word to CMD
00000001	Packet Data: WCFG Command
30002001	Type 1 Packet Header: Write 1 word to FAR
02000000	Packet Data: Frame Address = 0x02000000
3000401A	Type 1 Packet Header: Write 26 word to FDIR
00000000	Packet Data: word 1
...	
00000000	Packet Data: word 26
0000A53B	AutoCRC word

3.6.5 Virtex-II Pro Configuration Registers. The bitstream translation program must also account for the command words and settings written to each register that affects the status of the FPGA. Of the 15 configuration registers, only the CRC register, Frame Address Register (FAR), FDIR, Command Register, and Device ID register are written to in partial bitstreams.

Writing a group of configuration frames starts with a Type 1 packet header which indicates that 1 word will be written to the Command Register. The following word is the Write Configuration Data Command. This command word is followed by a Type 1 write header indicating 1 word is to be written to the FAR. The next word is the frame address formatted as shown in Figure 3.6. The following word is a Type 1 write instruction to write a specified number of words to the FDIR register. The FDIR is the register that the actual frame configuration data is written to. This sequence, shown in Table 3.1 is an excellent example of the commands needed to write to configuration memory.

The FAR is automatically incremented with every write to the FDIR or read from the Frame Data Output Register FDOR). Every time the address the FAR changes, the command in the Command Register is executed. For packets larger than $2^{11}-1$ words, the Type 1 packet can be immediately followed by Type 2 header which can specify higher word counts

Table 3.2: Bitstream Command Codes that Required Special Actions

Reset CRC	Set the calculated CRC value to 0
DESYNCH	Indicates the end of the bitstream. Program can exit.
IDCODE	Set BTP variables. No change to bitstream.

To ensure the bitstream being used for configuration was created for the target device, the FPGA configuration control logic requires that the correct device ID be written to the Device ID register. The BTP uses the device ID to determine the number of CLB, BRAM and BRAM interconnect columns in the FPGA.

The Write Configuration Data command and the correct Device ID must be written to the FPGA before writing to the FDIR register. These command are already part of the original bitstream and must remain unaltered. To translate a bitstream, only the frame address written to the FAR needs to altered since the remaining frames are addressed relative to the first frame in the packet. The method for calculating the new MJA is in Section 3.6.6.

3.6.5.1 Command Code Handling. Additional command codes that change the state of the FPGA configuration logic must be incorperated into the BTP. The codes that occur in partial bitstreams that require action are listed in Table 3.2. The remaining of command codes require no action must be and are left unchanged in the translated bitstream.

3.6.6 Calculating the New Major Address. Once the address being sent to the FAR has been identified, the Block Type and MJA is extracted from the frame address. Only the MJA will be altered, but the block type determines how it will be altered. Additionally, the architecture of the target Virtex-II (Pro) FPGA affects how the BTP calculates the new address. Using the number of CLB columns and BRAM columns in the target FPGA, the BTP performs error checking to verify that the address calculated targets an actual column within the reconfigurable area. The number of columns in each Virtex-II Pro device is listed in [Xil05b] and can be

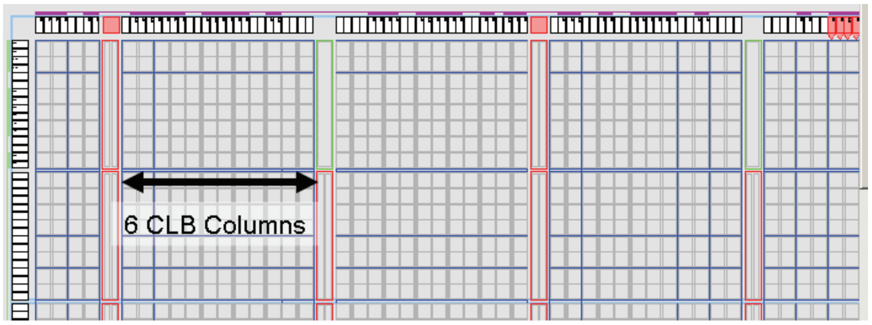


Figure 3.10: Distance between BRAM columns label in a view of the top left-hand corner of the XCV2P30 displayed in the PlanAhead.

automatically set by the BTP based on the Device ID read from the partial bitstream. The XC2VP30 has 46 CLB columns and 8 BRAM columns.

Critical to the calculation of BRAM addresses is the number of CLB columns between each BRAM/BRAM Interconnect column as illustrated in Figure 3.10. In all Virtex-II Pro devices there are 6 CLB columns between BRAM/BRAM Interconnect columns. Given the width of each reconfigurable module, *mod_width*, the width of the interconnect modules, *inter_width* and the number of CLBs between BRAM columns, *CLBs_between_BRAM*, the new MJA is calculated based on the current address and the number of modules the module is being moved as

$$new\_CLB\_MJA = old\_MJA + (dist\_in\_mods)(mod\_width + inter\_width), \tag{3.3}$$

$$new\_BRAM\_MJA = old\_MJA + \frac{(dist\_in\_mods)(mod\_width + inter\_width)}{CLBs\_between\_BRAM}. \tag{3.4}$$

A Block Type of 0, indicates that the MJA is a CLB column and the new MJA is calculated using (3.3). A Block Type of 1 or 2, indicates a BRAM column or BRAM interconnect column and (3.4) is used.

If interconnect modules are not used, *inter_width* is zero. Additionally, (3.3) and (3.4) could easily be altered to calculate the new MJA based on translation distance as in [KP06]. Specifying the translation distance allows more flexibility in the placement of the reconfigurable modules since they do not all have to be adjacent

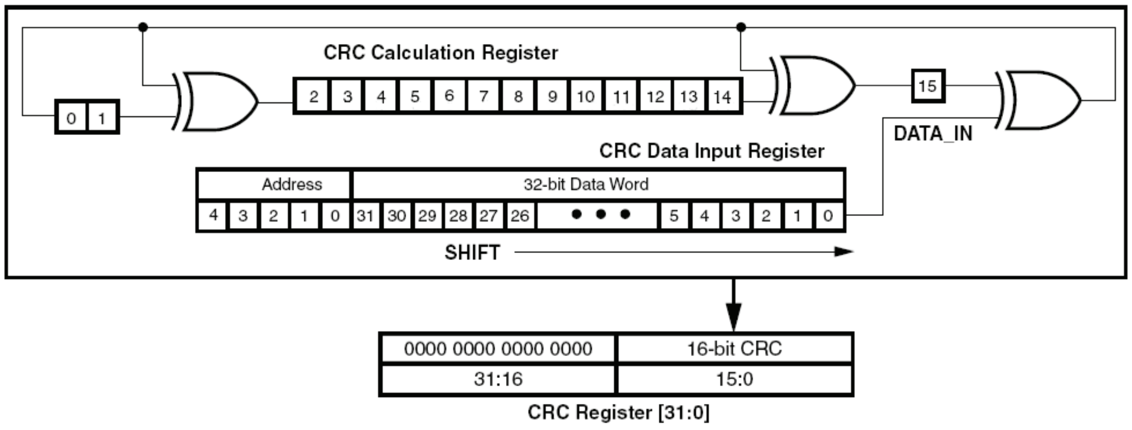


Figure 3.11: V2P Serial 16-bit CRC Circuity [Xil05b].

to one another. Since all of the proposed configurations use the modular design, translation distance is specified in modules for convenience of the user.

3.6.7 Updating the CRC Value. A 16-bit CRC verifies the integrity of the bitstream. For each 32-bit word written, with the exception of the Legacy Output register (LOUT), the CRC is updated. As shown in Figure 3.11, the 5-bit address code for the register and the 32-bit data word are used as input to the CRC calculation [Xil05b].

Also included in the CRC calculation is the Frame Address written to the FAR. Since this address is altered during the translation process, the original CRC value in the bitstream will not match the CRC calculated by the FPGA. To avoid a CRC error, the bitstream translation software must calculate the proper CRC value and alter the bitstream accordingly. Each time the bitstream translation software processes a 32-bit word that will be written to a resister other than the LOUT or CRC, the CRC value is updated.

In the Virtex-II Pro, CRC checks are performed in two different ways. In the first method, the CRC value is explicitly written to the CRC register using a Type 1 write packet header targeting the CRC register followed by the pre-calculated CRC value. This is referred to as an Explicit CRC. The second type of CRC check automatically takes place at the end of a write to the FDIR. Once the number of words indicated by the Type 1 or Type 2 read instruction have been written to the FDIR, the next word in the bitstream is a CRC value which is automatically written to the CRC register. This type of CRC check is referred to as an AutoCRC. The calculated CRC value is set to zero each time the CRC reset command is written to the command register or after the CRC register is written to using an Explicit CRC or AutoCRC with the current calculated CRC value.

To perform the CRC calculation in software an algorithm written for the Virtex series [Xil04d] was altered for the Virtex-II Pro and incorporated into the bitstream translation program. Changes included support for additional register designations and 5-bit register addressing. The code for updating the CRC can be found in Appendix B. As an option, bitstreams can be generated with CRC disabled. Before a partial bitstream with CRC disabled can be used to reprogram an FPGA, the FPGA must have been initially programmed using a bitstream with CRC disabled. The initial bitstream disables CRC checking on the FPGAs and all CRC values are expected to be set to 0x0000DEFC. The BTP has the option to disable CRC. If this option is set, the new bitstream uses the CRC values from the original bitstream (which should be 0x0000DEFC).

3.6.8 Overall Organization of the BTP. The high level organization of the BTP is shown in Figure 3.12. Since the CRC is not updated when a word is read from a register, no changes to the bitstream are necessary for a read packet. Since the frame address is written to the FAR using a Type 1 write before reading from the FDOR, the BTP can be used without modification to change the location that a bitstream will read from configuration memory.

3.7 FPGA Design Tools

To use a microprocessor in a partially reconfigurable design, both Xilinx Platform Studio and Xilinx Project Navigator are required. These two software suites are

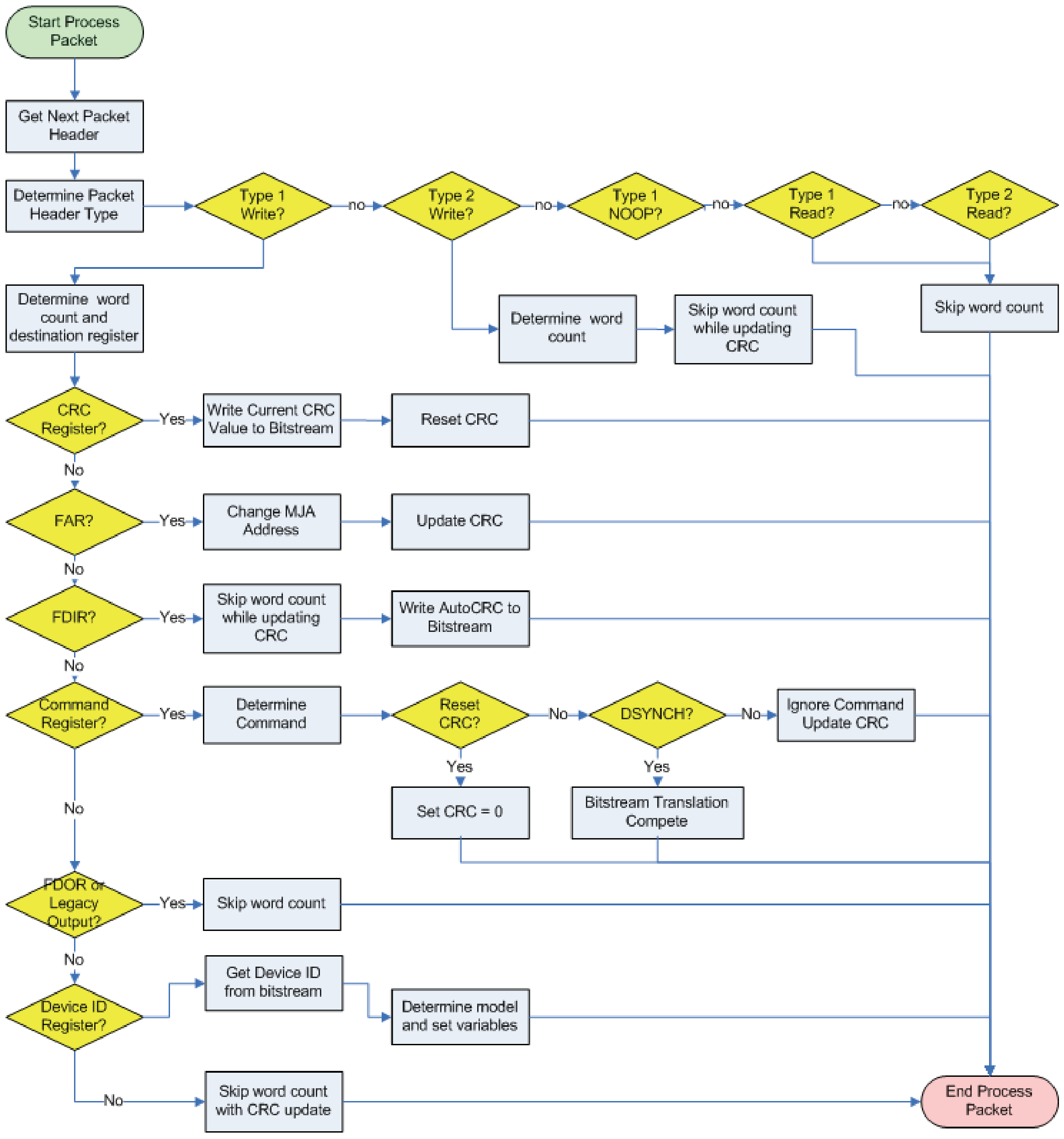


Figure 3.12: Flowchart for Processing a Packet. This flowchart shows actions for each packet header type. The program in Appendix B implements the above using a number of functions which decode command registers and commands to allow for debugging. Note all of actions that alter the bitstream occur in Type 1 or Type 2 write packets. All read packets are left unchanged in the translated bitstream. The CRC is updated for words written to all configuration registers except the LOUT. A valid partial bitstream will never write to the FDOR.

typically referred to by their major components the Integrated Synthesis Environment (ISE) and Embedded Development Kit (EDK), respectively. To perform partial reconfiguration, a supported version of ISE must be patched to include the appropriate version of partial reconfiguration implementation tools. The implementation tools are modified MAP, Place and Route (PAR), and library files that support partial reconfiguration [Xil07a]. The version of EDK that is used depends on the version of ISE that the implementation tools are available for.

Although implementation tools are available for ISE version 8.2i SP1, the design flow for implementing EDK projects changed dramatically between versions 8.1 and 8.2 of the tools. At this time, the of documentation explaining how to incorporate microprocessors developed in EDK 8.2 into partially reconfigurable designs created in ISE 8.2i SP1 is inadequate, so version 8.1 is used. All design and testing of the partial bitstream is conducted using ISE 8.1i SP1 with "PR_8" implementation tools (ISE 8.1.01i_PR_8_) and EDK 8.1 (Build_EDK_I.18.8).

PlanAhead provides a graphical environment for modular design and partial reconfiguration. PlanAhead version 8.2.6 is used to designate reconfigurable areas and to place bus macros. Although the constraints needed for modular designs can be created using the Floorplanner tool in ISE, PlanAhead provides a more intuitive and easier to use design environment. For reconfigurable designs that to do not incorporate a microprocessor, PlanAhead is used to build the partial bitstreams.

The partial reconfiguration support for PlanAhead greatly streamlines the generation of partial bitstream and eliminates the need for the rigid directory structure required by the traditional partial reconfiguration design flow. Unfortunately, at this time the PlanAhead partial reconfiguration flow does not include programming embedded microprocessors or the creation of the ACE file to load the FPGA configuration from a CompactFlash card.

For more control over the partial reconfiguration process, batch scripts are tailored for each design that uses a microprocessor. Custom batch scripts enable the process of building the partial bitstreams and creating a custom ACE file to be automated. An example script is included in Appendix C.

3.8 Implementing a Relocatable Partial Reconfiguration Design

The partial reconfiguration design flow for ISE 8.1.01i is well documented on the Partial Reconfiguration Early Access software tools web site [Xil07a]. The 8.1.01i partial reconfiguration design flow is not designed for relocatable modules and special considerations must be made when designing relocatable modules.

3.8.1 Reconfigurable Modules. Relocatable modules are reconfigurable modules that can be moved on the FPGA. As part of the User Constraints File (UCF), which defines logical constraints such as pin connections, instances of VHDL components can be given area constraints to restrict the area they will be implemented in. These area groups can be designated reconfigurable regions. Each reconfigurable module is defined as its own reconfigurable region. Although a frame in a partial bitstreams reprograms a fraction of a column, the partial reconfiguration design flow for ISE 8.1.01i allows reconfigurable regions to be defined as any rectangular size. Thus, for reconfigurable modules in Virtex II series devices no longer have to be the full height of the column [Xil06]. The partial bitstream produced reprograms an entire column if any portion of the frame in the reconfigurable area changes. Since the resources available on the XUPV2P are not homogenous between columns (cf., 3.5), the ability to specify shorter reconfigurable modules allows the logic used by each reconfigurable module to be targeted to the portion of the column that has consistent resources across the FPGA. Figure 3.13 shows the areas that have homogenous resources as rectangles above and below the PowerPCs.

3.8.2 Bus Macros. All signals entering or exiting a reconfigurable region must be routed through bus macros. Bus macros connect two reconfigurable regions together or connect a reconfigurable region to a static portion of the design. Bus macros define intermodular routing, forcing the interconnections of different reconfig-

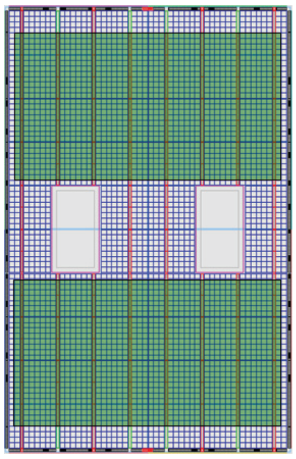


Figure 3.13: Areas with Homogenous Resources Across Columns. Restricting the reconfigurable area to the portions of the FPGA that have consistent resource across all columns ensures that the resources needed by module is available when relocated. The regions shown above avoid the MGT, PowerPCs and I/O banks.

urable modules to be identical. This insures the reconfigurable modules have identical interfaces. Bus macros are hard-placed and hard-routed. The bus macros compatible with ISE version 6.x are available as part of XAPP290 [Xil04c] and versions compatible with ISE 8.x are available from the Early Access Partial Reconfiguration lounge [Xil07a].

Partial reconfiguration for ISE 8.x allows static routes inside the partially reconfigurable regions [Xil06]. This is a change from the requirements of XAPP290 [Xil04c], which specifies that *all* signals must be routed through bus macros. This change simplifies the design of Partial Reconfiguration design with non-relocatable modules and improves timing. In non-relocatable designs, different version of the modules are created by the partial reconfiguration tools. Partial bitstreams include the unique functionality of that version and the static routing at that location. Since the static routing is included in all versions of the reconfigurable module, static routing remain intact during reconfiguration. However, relocatable modules static routes within reconfigurable modules must be prevented since reprogramming a module with a bit-

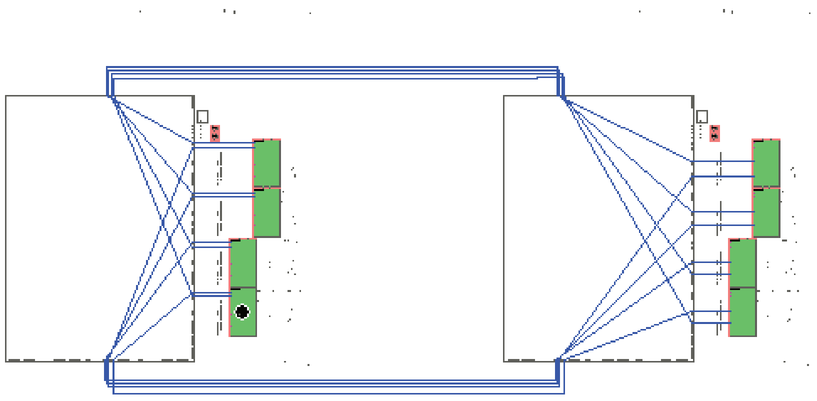


Figure 3.14: The Layout of a Bus Macro in FPGA Editor.

stream that does not program the static route will destroy any static routes that pass through the module.

3.8.3 Making Reconfigurable Modules Relocatable. If static components are placed on the FPGA and the reconfigurable modules are relocated, static logic must not be allowed to use any resources within the reconfigurable areas. Static routing in reconfigurable modules can be prevented by using the XIL_PRCONTROL file to override the default behavior of the partial reconfiguration design flow. The Allow_Routing_In_Dynamic_Area field prevents the static portions of the design from routing through the reconfigurable regions.

Using area constraints to confine a reconfigurable module to a subregion of the column-based module, as explained in Section 3.8.1, ensures resources needed by the module are available at all locations but allows static routes to use the resources above and below the reconfigurable area. In order for Allow_Routing_In_Dynamic_Area to prevent any static routes from being programmed by the partial bitstream, the reconfigurable module must span the entire height of the column. For systems with static logic, this further reduces the number of locations for relocatable reconfigurable modules.

Using Allow_Routing_In_Dynamic_Area severely limits the number of locations that a reconfigurable module can be placed. In addition to resources such as DCMs and BUFMUXs which must have routing resource available, designs using the PowerPC require connections to the BRAM interconnect columns around the PowerPC core as well as a connection to the PowerPC JTAG. Thus, if connections to unique resources must pass through the reconfigurable regions, using Allow_Routing_In_Dynamic_Area effectively prevents the router from making necessary connections.

3.9 Internal Reconfiguration

To support internal partial reconfiguration, a microprocessor feeds the partial bitstream to the HWICAP. The microprocessor can also translate the partial bitstreams. To support internal partial reconfiguration, the design includes static and reconfigurable regions. The static regions contain the microprocessor, peripherals and other portions of the circuit that will not be altered during reconfiguration.

3.9.1 Using an Embedded Microprocessor to Run the BTP. BTP was developed and testing on a PC using Microsoft Visual Studio 6.0. To test the BTP, bitstreams created by the Xilinx partial reconfiguration tools were translated and used to program the FPGA through the JTAG. Since partial bitstreams are the same no matter which configuration method is used, the same bitstreams can be used to program the FPGA through the ICAP.

To actively restore redundancy on an FPGA, the microprocessor can translate a stored bitstream and reprogram the FPGA. The Virtex-II Pro supports both MicroBlaze and PowerPC microprocessors processors. Each of these microprocessors supports internal reconfiguration through the ICAP so either could be used to translate partial bitstreams, relocate a module, and send the partial bitstream to the ICAP interface. Each processor has advantages and drawbacks.

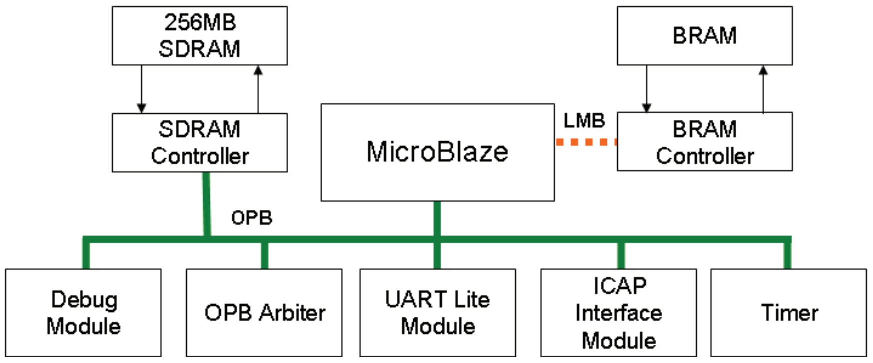


Figure 3.15: MicroBlaze System Block Diagram. The connections between the MicroBlaze and its peripherals are shown. The MicroBlaze uses a Local Memory Bus (LMB) to access BRAM, but uses the OPB to connect to all other peripherals.

3.9.2 MicroBlaze and uClinux. Presumably, the microprocessor on the FPGA will be used for other purposes until it is needed to reconfigure the FPGA to recover from a fault. To show that the bitstream translation program can be run on the microprocessor along with other applications, the BTP is run on uClinux. uClinux is a port of Linux designed to work on microcontrollers that do not have Memory Management Units. A benefit of using uClinux with the MicroBlaze is availability of ICAP drivers [WB04]. These drivers take the low level drivers written by Xilinx and “wrap” them so the ICAP can be mounted as a device in the operating system and accessed using build-in Linux device commands. The block diagram for the MicroBlaze design is shown in Figure 3.15. The requirements for using uClinux on the MicroBlaze can be found in [WSW06]. Instructions for using the ICAP in custom program can be found in [WB04]. The BTP program only requires minor changes to work in uClinux to account for the reversed byte order in uClinux *fread()* and *fwrite()*.

One drawback of using the MicroBlaze is the additional area needed to instantiate the microprocessor. Since the PowerPCs are already on the FPGA, not using them is waste of reconfigurable area that could be used to increase the number of locations for spare modules. Furthermore, in addition to the MicroBlaze, all necessary peripherals including the HWICAP, memory controller, OPB controller, UART, timer, debug module, and DCMs must be instantiated.

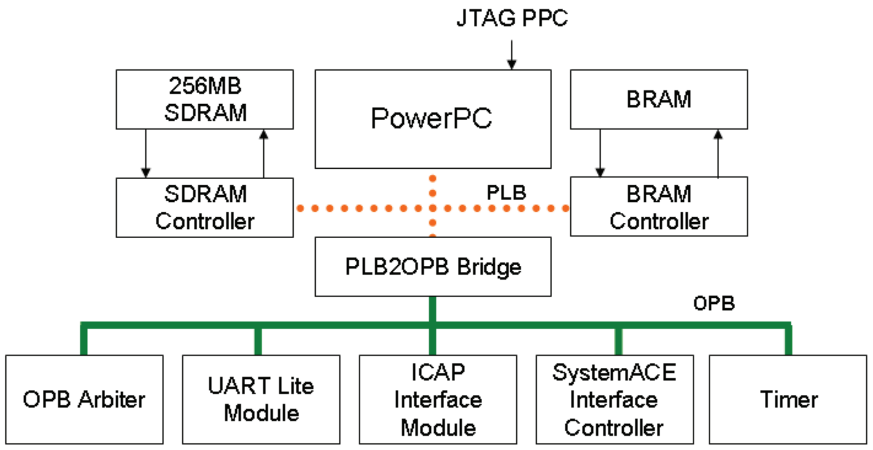


Figure 3.16: PowerPC System Block Diagram. The connections between the PowerPC and its peripherals are shown. The PowerPC uses a Processor Local Bus which must be bridged to access peripherals on the OPB.

3.9.3 PowerPC. The XUPV2P has two PowerPC 405 cores embedded in the FPGA. Although many of the same peripherals used on the MicroBlaze need to be instantiated for use with the PowerPC, the actual microprocessor does not use any reconfigurable area on the FPGA with the exception of a small number of configuration bits stored in adjacent BRAM interconnect columns. Xilinx's ICAP drivers can be used to write bitstreams to the ICAP. Figure 3.16 is the block diagram of the PowerPC with the necessary peripherals to take a partial bitstream from a compact flash, store it in memory, translate the bitstream and reprogram the FPGA through the ICAP. Instructions on how to set up a PowerPC for partial reconfiguration are found Appendix B. The BTP for the PowerPC can be found in Appendix B.

Like the MicroBlaze, the PowerPC as some drawbacks. In FPGAs with multiple PowerPCs, both PowerPCs must be connected to the PowerPC JTAG chain. Unless both PowerPCs are instantiated in the design and properly connected to the jtagppc_cntlr signal, the EDK design will not pass EDK design rule checks. The existence of the non-global jtagppc_cntlr signal which needs to connect to the otherwise unused PowerPC core requires static logic be routed through the reconfigurable area around the PowerPC.

3.10 Summary

This research determines if an efficient fault recovery system can be developed which allows a user circuit to operate through fault. Efficiency is gained by not storing individual bitstreams that implement the same function multiple locations on the FPGA. Instead, the bitstreams are relocated by the BTP. The BTP is evaluated to determine if successfully translates bitstreams. The suitability of using the XUPV2P to implement the fault recovery system is also determined.

Additionally, this research explores using relocatable modules to implement fault tolerant designs in FPGAs. The designs take advantage of portions of the Virtex-II Pro that have homogenous resources to make the modules relocatable by simply changing the frame addresses in the partial bitstreams. Each configuration will be tested to validate that the designs work.

IV. Implementation

4.1 Introduction

The primary goal of this research is to develop an efficient fault recovery system that allows a user circuit to operation through faults. Three TMR configurations are developed. Each of these configurations can provide passive redundancy and support the replacement of modules without interrupting the correct operation of the user circuit. The BTP correctly translates the partial bitstreams and can be implemented on an embedded microprocessor to perform internal partial reconfiguration. Additionally, the XUPV2P development board provides sufficient resources to implement the fault recovery system and user circuit.

4.2 Verifying Relocation of Partial Modules

The changes made by performing partial reconfiguration must be verified. Since all changes are internal to the FPGA, the only way to determine if the modules are relocated properly is to design the circuit so that the effect of the reconfiguration is evident. A straightforward design to test is the interconnect module design shown in Figure 3.3. Modules 1, 2, and 3 are programmed to add 1, 2 and 3, respectively, to the input received on the input bus. All other functional modules pass data through the bus macros unchanged. The results from the functional modules are placed on their corresponding data busses by connecting through the interconnect modules as shown in Figure 4.1. The 4 LSB of the data received on each result bus and the input bus is displayed in hexadecimal on four 7-segment displays. Each of the 7-segment displays correspond to one of the result busses or the data input bus as shown in Figure 4.2.

Figure 4.2a shows the initial configuration with modules 1, 2, and 3, connecting through interconnect modules to busses 1, 2 and 3 respectively. Figure 4.2 shows the results of the partial bitstream for module 1 being translated and used to replace module 2. Similarly, the results of routing changes can also be observed using this configuration.

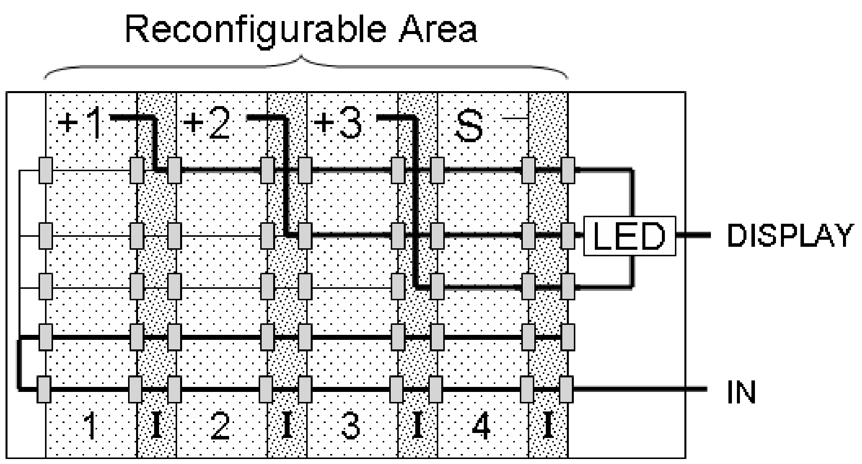


Figure 4.1: Interconnect Module Layout Test Configuration. To determine if module translation is successful redundant functional modules are replaced with modules that add 1, 2 or 3 to the input. A 7-segment display is used to display the results. The spare modules is labeled with "S".

Before implementing the modular designs developed in Chapter III on an FPGA with a microprocessor, each configuration is tested by translating the partial bitstreams using the BTP running on a PC and reprogramming the FPGA using the JTAG. This technique determines if the translation and reprogramming of the module is successful.

4.2.1 Testing the Interconnect Module Designs. Testing the interconnect module design requires bitstreams for the functional module and each of the interconnect modules. To generate these partial bitstreams, the configuration shown in

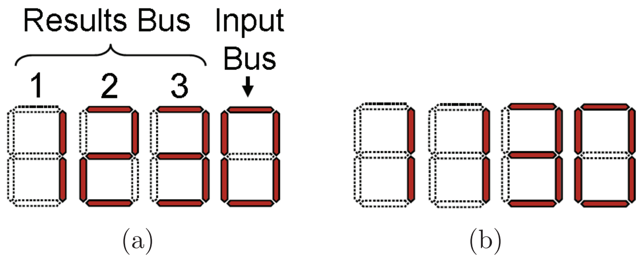


Figure 4.2: Partial Reconfiguration Status Display. The test modules and top level layout are designed to show which module connects to each bus.

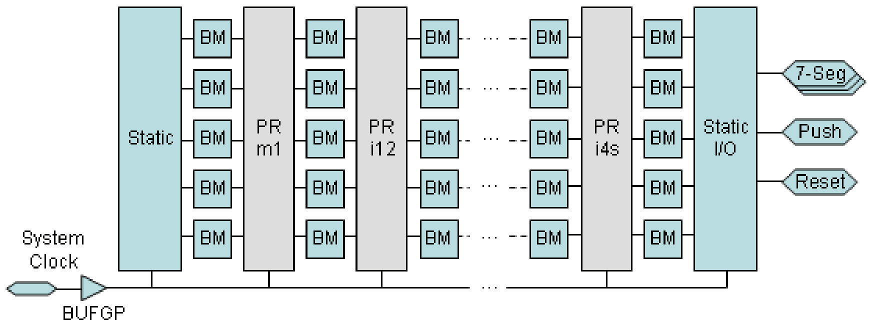


Figure 4.3: Top Level VHDL Organization.

Figure 4.1 is implemented. The configuration uses 3 active functional modules, 1 spare functional modules, and 4 interconnect modules. The top level design, shown in Figure 4.3, contains all I/O instances, clock buffers, base design instantiations, partial reconfiguration module instantiations, signal declarations and bus macro instantiations needed for the design in accordance with [Xil06]. Each of the static and reconfigurable modules are created and synthesized separately.

4.2.1.1 Setting Design Constraints Using PlanAhead. To avoid the nonhomogeneous resources on the FPGA, the design is placed in the lower right corner of the FPGA below the PowerPC as shown in Figure 4.4. All module constraints and bus macros placement are set using PlanAhead. PlanAhead also performs design rule checking to verify that all design rules have been met for partial reconfiguration. The process of laying out the design in PlanAhead uncovered a major flaw in the use of interconnect modules. According to design rules, the minimum width for a reconfigurable area with bus macros placed on both sides of the reconfigurable area is two CLB columns. Since very little logic is implemented by the interconnect modules, and no other logic is allowed in the region, using two entire columns for routing prevents valuable resources from being used.

The configuration shown has 4 CLB wide functional modules and 2 CLB wide interconnect modules thus, approximately 33% of the CLB resources the reconfigurable region is used by the interconnect columns. Considering that the only purpose of the interconnect modules is routing, this may be an unacceptable use of resources.

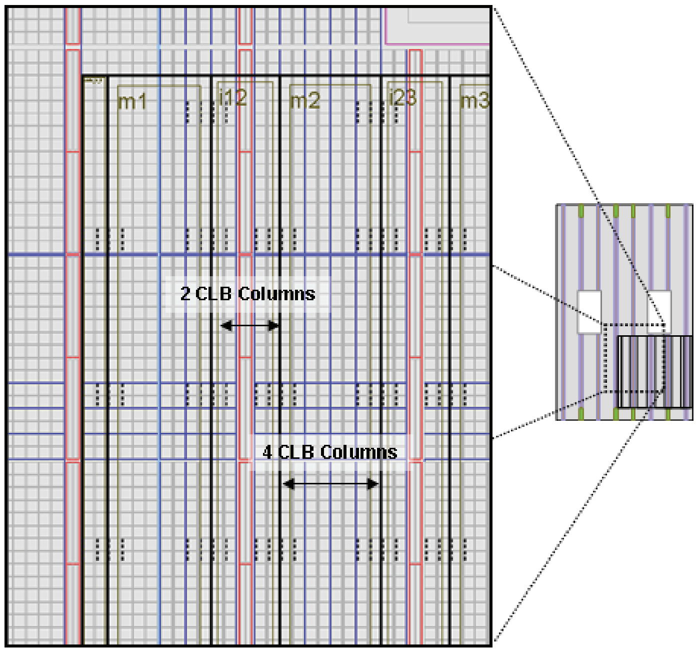


Figure 4.4: PlanAhead Layout for the Interconnect Module Configuration. A portion of the area constraints of the reconfigurable modules are shown. Bus macros can be seen along the right and left sides of the reconfigurable regions.

Although this configuration shows 100% of the BRAM/BRAM Interconnect columns in the reconfigurable region being used by the interconnect columns, this was only for convenience of module placement and could be used by the functional modules.

4.2.1.2 Translating the Modules. To test the BTP each of the four functional module are translated to target each of the other functional module locations. The bitstreams produced by the BTP program the FPGA and their effect is determined by the status of the system shown on the 7-segment display. The circuit works as expected except when bitstreams for modules 2, 3, 4 are translated to target the location of module 1. Using a translated partial bitstream to reprogram module 1 causes the circuit on the FPGA to crash. The result is that only one digit on the 7-segment display remains lit.

The 7-segment display is a common anode display which uses the same anodes to drive all 4 digits on the display. Displaying only one digit is an indication that

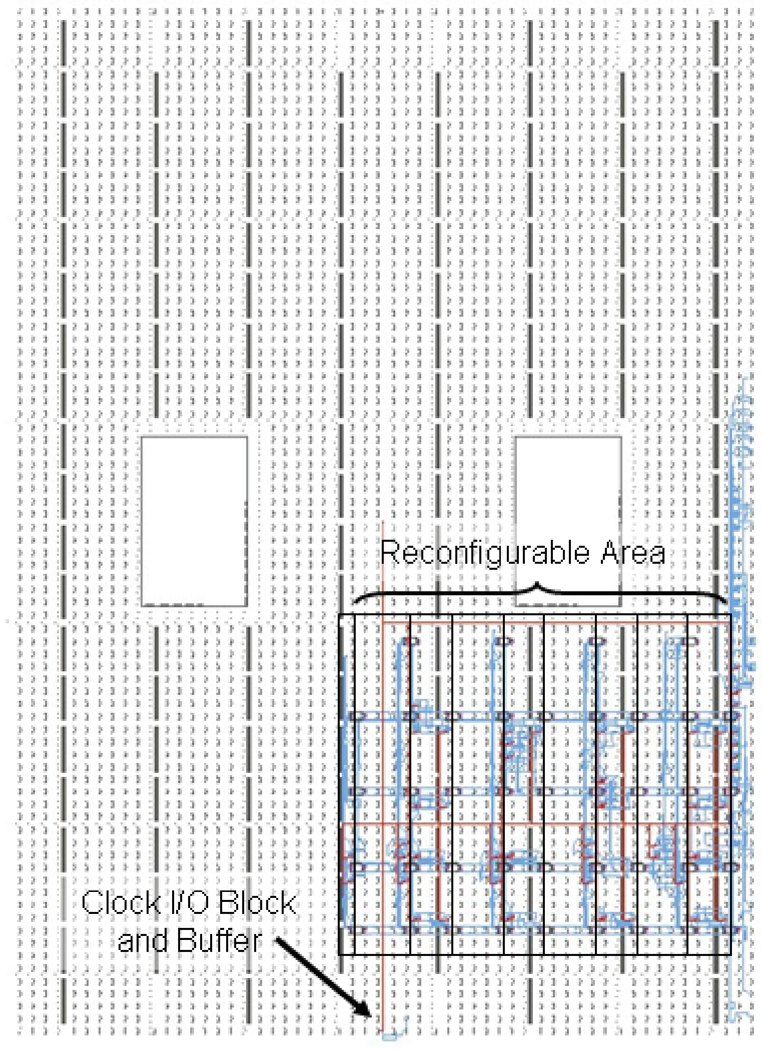


Figure 4.5: FPGA Editor View of NCD for Interconnect Modules Design. The reconfigurable modules are outlined in black. The buffered clock signal that is distributed to all reconfigurable modules and the 7-segment display driver in the static logic is highlighted. The BUFMUX connection is in the same column as module 1.

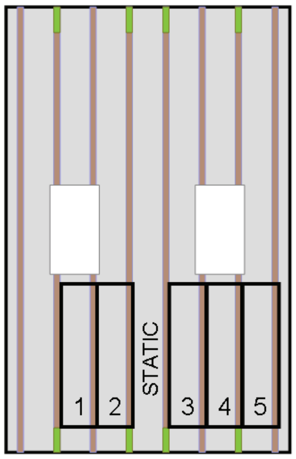


Figure 4.6: Area Constraints Direct Connect and LUT-based Configurations. To avoid disconnecting the system from the clock, the columns that include the I/O blocks for the system clock are not using in the reconfigurable modules.

the clock driving the state machine to determine which of the 4 digits should being drive has been disconnected. By examining the NCD file in FPGA editor, shown in Figure 4.5, it can be seen that static routing in the same column as module 1 connects the system clock input to the BUFGMUX for clock distribution. Moving functional module 2, 3, or 4 to location 1 removes the connection to the buffered system clock and reprograms the I/O block. To prevent the system from crashing, modules that have static routing can not be written over using translated bitstreams from another location. However, in the configuration developed in Chapter III, the columns that include connections to the clock pins and for routing the output of the BUFGMUX can be used as one of the initial locations of the modules.

To test the interconnect modules, each of the interconnect modules were translated to each of the other interconnect module locations. The circuit performed as expected for each reconfiguration; no problems were observed.

4.2.2 Testing the Direct Connect Modular Design. To test the direct connect module design, a top level-design based on the layout in Figure 3.1 is developed using functional modules that add 1, 2, and 3 to the value on the input bus. Modules 4 and

5 pass input and data bus values without changing them. A diagram of the top level design is shown in Figure 4.7. Interconnect modules are not used in this layout and the functional modules must be altered, and resynthesized to connect directly to result busses. That is, each of the active functional modules replace the data on one of the results busses with the result of the function it implements. Once again PlanAhead is used to layout the reconfigurable area and produce partial bitstreams. The layout for the reconfigurable modules is shown in Figure 4.6. It avoids static routing and clock pin connections in the center of the FPGA by creating a gap between reconfigurable modules where static logic can be placed. Since functional modules implemented by the partial bitstreams connect to specific busses, the design is more difficult to test.

To verify that the partial bitstreams reprogram the FPGA properly, the partial bitstream for module 4, which only passes through the data is used to "remove" all of the modules. When modules are "removed" the display reads "0000" since the modules in all locations place the data they receive on the inbound result busses onto the outbound result busses without changing the data. Translated bitstreams for modules 1-3 are used to implement functional modules in locations other than their original locations. Since the proper translation of the module 4 bitstreams is confirmed by their successful removal to the original modules, they can be used to verify that other bitstreams are translated properly. Using the module 4 bitstreams, the target of translated bitstream can be verified by replacing it with the module 4 bitstream targeted to the same location.

4.2.3 Implementing the LUT-based Modular Design.

The LUT-based modular design, shown in Figure 3.4, requires additional constraints to be placed on the modules for synthesis and implementation. Care must be taken to ensure the LUTs used to control the multiplexors are not broken into multiple LUTs or eliminated due to optimizations during implementation. To test the concepts behind the LUT configuration, a simple design with three LUT controlled multiplexors is created using the LOC, BEL, and LOCK_PINS constraints. The LUT and BEL constraints force

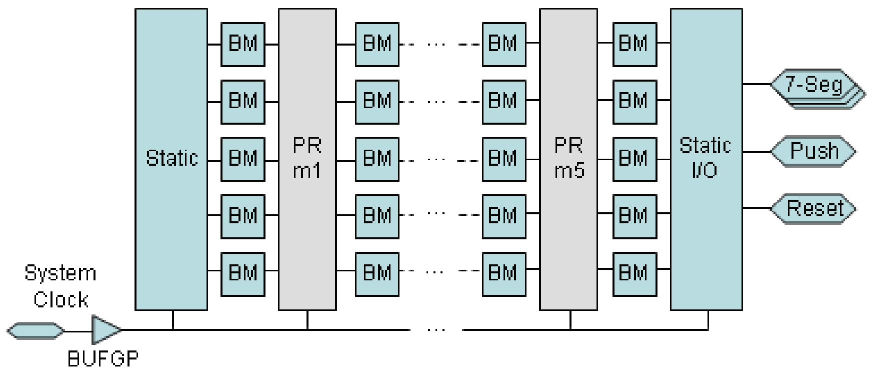


Figure 4.7: Top Level VHDL Organization for the Direct Connect and LUT-based Designs.

the three LUTs to be implemented in specific locations and LOCK_PINS prevents the pins of the LUT from being switched [Xil05a]. Forcing the LUTs to be in the same column, not only makes them easier to find, it also minimizes the number of frames changed by the difference-based partial bitstream. Since the locations of the LUTs are known, the NCD file is easily edited to invert the equations implemented by the LUTs.

Producing the difference based partial bitstream using the BitGen utility, as described in [Xil04c], produces a bitstream which changes 49 frames. This is higher than expected since an entire CLB column is only 22 frames [Xil05b]. The BitGen utility takes a .NCD file and compares it to an existing bitstream to determine which frames changed. Using the BitGen utility on the original .NCD file reveals that 48 frames have changed. Using the BTP to analyze the bitstreams, it can be seen that both bitstreams change a series of frames with MJAs equal to 3, 4, 47, 48, and 49, but only the partial bitstream produced from the altered .NCD file programs a frame with a MJA of 35. This MJA corresponds with the location of the LUTs.

Although PlanAhead identifies the LUTs in the .NCD file for the reconfigurable modules and permits the addition of location constraints, it does not include these constraints in the user constraint file it creates for generating the partial bitstreams. Only top level constraints are included. Even when the constraints are manually added to the .UCF file used by the NGCBuild process for each reconfigurable module, the LUTs cannot be constrained and are optimized out of the design. Due to this limitation in the partial reconfiguration design flow, a reconfigurable design which changed the routing within a reconfigurable region could not be demonstrated.

4.3 Adding a Microprocessor to the Design

Adding a microprocessor to the same FPGA as the reconfigurable circuit increases the complexity of the design. For testing, the microprocessor is added as a module of the top-level design and has no direct connection to the reconfigurable area. Assuming the location of the fault is known, the microprocessor on the FPGA can perform reconfigurations to repair the fault. In an actual system the reconfigurable area would be connected to the microprocessor to report faults that have been detected.

4.3.1 Resources Used By Microprocessors. To evaluate the amount of resources needed by each microprocessor and the peripherals used in the configurations in Figures 3.15 and 3.16, the designs are synthesized in EDK and exported to PlanAhead. In a typical FPGA design, the number of resources needed to implement the design is a good indication of how much of the FPGA the design will use because unused resources can usually be used by other components in the system. However, in a modular reconfigurable system only static modules can be placed in the same region as the microprocessor and peripherals. PlanAhead is used to determine how large the module containing the microprocessor and peripherals must be to provided the required resources. In both cases, the modular areas needed to be expanded until they included the required amounts of BRAM.

Resources needed by the MicroBlaze and PowerPC with the peripherals shown in Figures 3.15 and 3.16, are shown in Table 4.1. Although the PowerPC uses more resources, this is largely due to the fact that the PowerPC design used 64KB of BRAM, compared with only 8KB of local memory for the the MicroBlaze design, and includes SystemACE loading of the initial configuration and access of partial bitstreams on

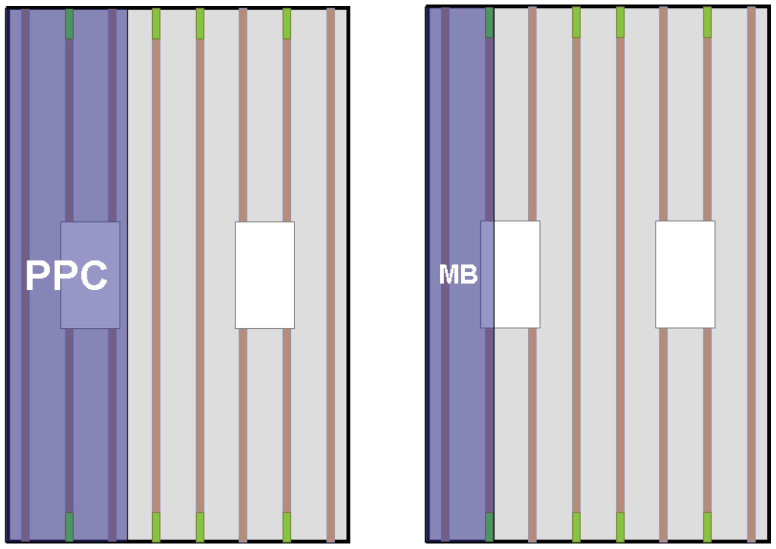
Table 4.1: Resources Required to Implemented each Microprocessor and Peripherals.

	PowerPC			MicroBlaze		
Resource	Available	Required	Utilized	Available	Required	Utilized
LUT	8,576	3,459	40.33%	4,922	3,204	64.18%
FF	8,576	3,078	35.89%	4,922	2,382	47.72%
SLICE	4,288	2,110	49.21%	2,492	1,954	78.29%
MULT18X18	50	0	0%	34	3	8.82%
RAMB16	50	33	66%	34	29	85.29%
TBUF	2,144	0	0%	1248	0	0%

a compact flash card. The large difference in BRAM usage is due to the difference in operating systems and storage of the BTP, and are not inherent to the choice of microprocessors. Figure 4.8 shows the portions of the xc2vp30ff896-7 that must be reserved for each processor and peripherals.

Although the MicroBlaze has a smaller footprint than the PowerPC, the PowerPC was chosen over the MicroBlaze due to stability problems encountered during implementation. Additionally, the design flow for the MicroBlaze required a custom uClinux kernel be compiled for each major hardware and minor software change, making testing cumbersome. The PowerPC design used a Xilinx standalone operating system which could be quickly recompiled in EDK. The operating system used on the PowerPC could also be used on the MicroBlaze, but it was not expected to improve stability.

4.3.2 Changes to BTP for PowerPC. The BTP program requires changed to use drivers written for the PowerPC and, to eliminate the need to load the BTP into external memory, to place into the 64KB of BRAM used as internal memory. To reduce the size of applications written for Xilinx embedded microprocessors, XPS includes Xilinx versions of standard C libraries. Although they have less functionality, they also use less memory than the standard C libraries. XPS automatically adds libraries for standard C functions so care must be taken to avoid using common



(a) (b)

Figure 4.8: Footprint for each Microprocessor and Peripherals. The portion of the xc2vp30ff896-7 FPGA that must be dedicated to the PowerPC and peripherals (a) and the MicroBlaze and peripherals (b).

functions as a printf(), which causes the stdio to be included in the compiled version of the BTP increasing its size beyond 64KB.

On startup, the partial bitstreams are loaded from the compact flash into extended memory. To minimize the memory used to store bitstreams, BTP translates the bitstream in memory without copying it. Therefore BTP stores the location the bitstreams are currently targeted to and calculates new MJAs based on the desired target.

To verify that the PowerPC version of the BTP correctly translated the bitstreams, the CRC values are calculated are compared with those calculated for the same bitstreams using the PC version.

4.3.3 Internal Reconfiguration using the PowerPC. To test the PowerPC and make sure it properly applies the partial bitstreams to the ICAP, the blanking bitstreams generated by the partial reconfiguration tools are used. Blanking bitstreams contain all of the static routing and logic within the reconfigure module, but not the logic implemented by the reconfigurable module. Since the bus lines that pass through the results are part of the reconfigurable module, using the blanking bitstreams removes these connections. The digits shown on the 7-segment displays were consistent with the connections being removed and the circuit could be repaired by reprogramming with the original bitstreams for each module.

4.4 Preventing Static Routing

To allow modules to be relocated internally using the PowerPC, static routing in dynamically reconfigurable areas must be prevented. The PowerPC requires dozens of I/O pin including clocks, external memory, compact flash and the UART. These I/O pins are located on all sides of the FPGA package so static routing in dynamic areas can not be avoided just by placing the reconfigurable portion of the circuit in a certain area of the FPGA. Figure 4.9 shows all the routing on the FPGA for the interconnect module design with a PowerPC and peripherals. The NCD file was created with no restrictions on static routing in reconfigurable areas.

Although restricting the location of the reconfigurable areas to the area below the PowerPC ensures all of the resources available at each location are homogenous, it limits the effectiveness of the Allow_Routing_In_Dynamic_Area override. The override is used to prevent static routing in the dynamic area, but does not prevent routing above and below the reconfigurable area. The static routing above and below the area will be included in the partial bitstream. To prevent static routing from being included in the bitstream, the reconfigurable module must be frame bounded. That is, the reconfigurable area must include entire frames. In the case of the Virtex-II Pro, the reconfigurable modules must span the entire column.

4.4.1 Problems Caused by Restricting Routing. If static routing within reconfigurable areas is not allowed, design errors occur when unique resources utilized by the static logic fall within the reconfigurable areas. Full column height reconfigurable areas in the Virtex-II Pro make it difficult to avoid including unique resources

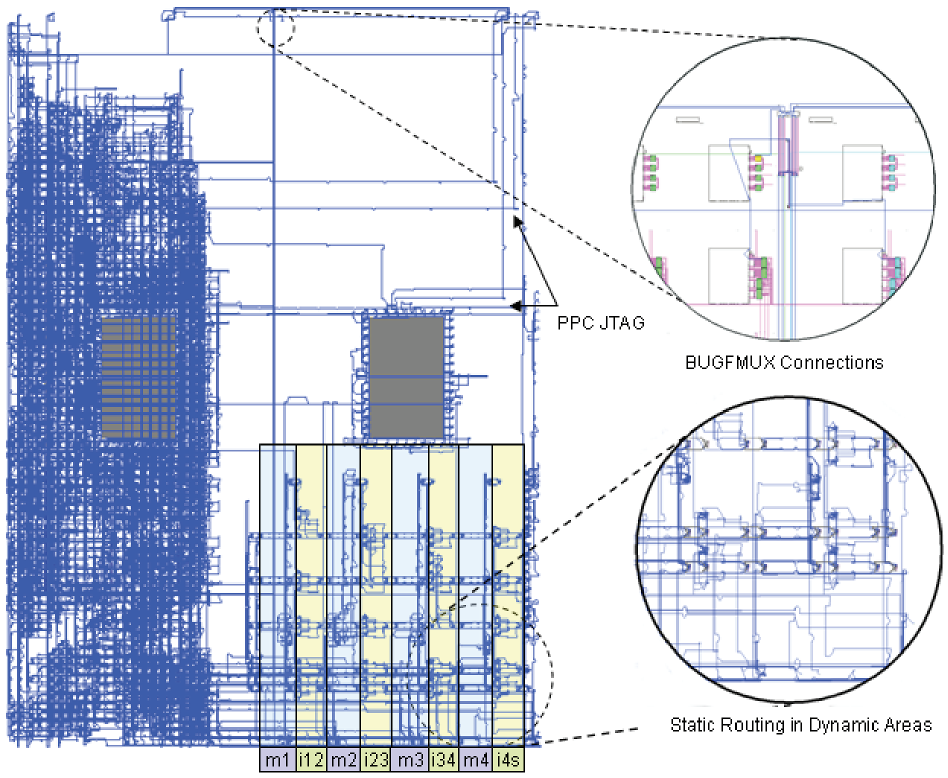


Figure 4.9: FPGA Editor View of Interconnect Module Design with the Power PC. The major components and all routing used for this configuration are shown here. When relocating a module, all of the logic and routing programmed by the configuration bitstream in the columns it occupies will be relocated. In additional to the routing around the righthand PowerPC, including the PPC JTAG, the routing connection made to distribute the output of the BUGFMUXs and the static routing crossing through the reconfigurable areas are highlighted. Also note the bus lines that make connections above the boundaries of the reconfigurable areas.

in reconfigurable area and prevent some of the areas from being used. Prohibiting static routing within the reconfigurable areas prevents the resource within the reconfigurable area from being connected to the static logic. This problem does not exist in non-relocatable modular partial reconfiguration since each version of the reconfigurable module includes the necessary static logic.

4.4.1.1 Unique Resources. In the VC2V2P30, the BUFMUXs located at the top and bottom of the center CLB columns of the FPGA can not be accessed by static logic if these CLB column are included in the reconfigurable region. BUFMUXs are used to buffer clock inputs and must be in designs that include a microprocessor. Although, if static routing is not allowed, connections to DCMs can also become unroutable if the DCM being used is in a reconfigurable area. On the VC2V2P30 there are 8 DCMs and only two DCMs are required for the PowerPC or the MicroBlaze designs. Location constraints must be used to ensure that the DCMs used are in the static area.

Using the PowerPC to perform partial reconfiguration adds additional constraints to the areas where static routing can safely be restricted. The VC2V2P30 has two PowerPC cores. Although only one of the PowerPCs is used for the BTP and to reprogram the FPGA, the other PowerPC must be connected to the global ground signal and the PowerPC JTAG. EDK does not allow the second PowerPC to be removed from the design because, by design rules, both must be connected to the PowerPC JTAG. The routing around the second PowerPC is clearly evident in Figure 4.9.

4.4.2 Programming of I/O Blocks. Configuration data to program the I/O blocks at the top and bottom of the FPGA is included in the configuration frames [Xil04b]. If any pin connections are made within the configuration frame, moving the frame will presumably create new connections to the corresponding I/O blocks at that location. Although these connections may not affect the function of the circuit, using a translated bitstream to replace a module that makes pin connections will destroy

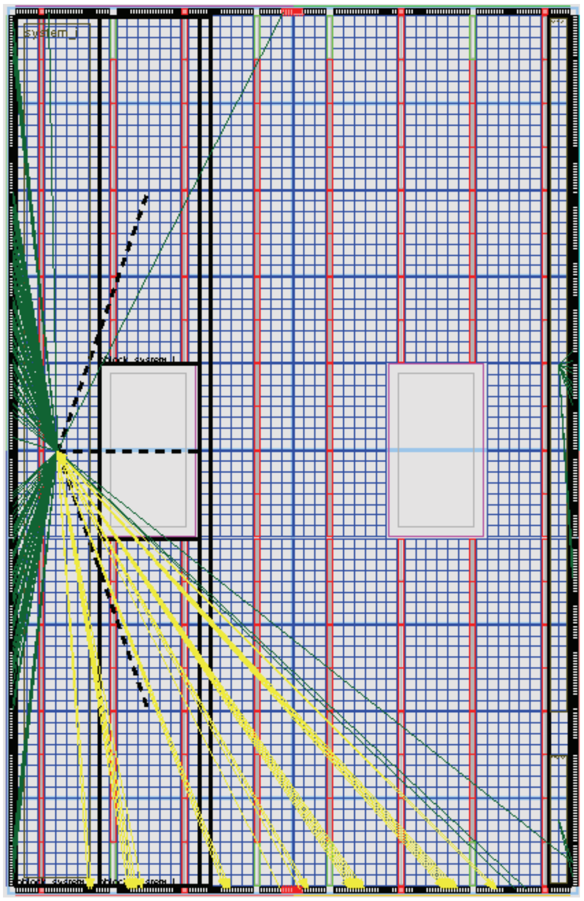


Figure 4.10: I/O Blocks Used by Static Logic. The PowerPC and display logic connect to I/O blocks located throughout the FPGA. The SystemACE Compact Flash connections are highlight with a lighter color (yellow).

all of the connections made by the original module. Additionally, it is unknown if permanent damage can occur from configuring I/O blocks improperly.

On the XUPV2P, the I/O blocks for external memory are located on the left side of the FPGA and the SystemACE Compact Flash connections are on the bottom of the FPGA. Figure 4.10 is a view from PlanAhead with only the PowerPC and static logic placed. The I/O block connections are represented by lines radiating from these modules. The connections on the left and right side of the FPGA do not cause a problem in the column-based relocatable designs, but connections on the top and bottom of the FPGA prohibit relocatable modules from being placed in that location. The SystemACE connections are shown highlighted using a lighter color (yellow).

In addition to the SystemACE, the DDR clock output and UART connect to the top of bottom of the FPGA. Although it makes testing more difficult, the SystemACE can be removed and partial bitstreams can be loaded directly into memory using the Xilinx Microprocessor Debugger (XMD). Although in a real system it would be possible to removed the UART, it is not practical in a development system. The DDR clock output is in the static region so it does not cause a problem.

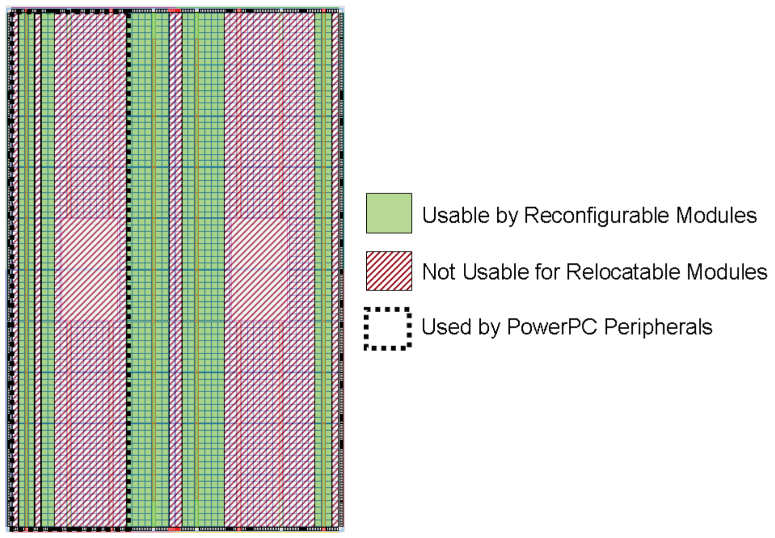
4.5 Safe Locations for Relocatable Modules on the XUPV2P

Removing the SystemACE Compact Flash decreases the number of columns that are effected by I/O block placement, but the number of locations where relocatable modules can be placed on the XUV2P is still severely limited. Figure 4.11 shows the locations reconfigurable modules can and can not be when using a PowerPC to run the BTP with support for the UART.

Given the minimum width of each reconfigurable region is two CLB columns, the number of reconfigurable regions can be no greater than 6. If BRAM is required by the relocatable modules, the number locations for the module is reduced to 3.

4.6 Errors During Bitstream Generation

By restricting the relocatable modules to the areas determined to be safe as shown in Figure 4.11, a simple proof of concept design with only two relocatable module can be created to demonstrate that the PowerPC can relocate the two modules and reprogram the FPGA. The design is assembled properly when static routing is allowed in the dynamic area but when static is prohibited, the process for creating the partial bitstreams fails during the merge phase. In the merge phase the complete design is built from the base design, containing the static logic, and each of the reconfigurable modules.



Usable by Reconfigurable Modules
Not Usable for Relocatable Modules
Used by PowerPC Peripherals

Figure 4.11: Location Suitability for Relocatable Modules. Considering required I/O blocks, PowerPCs, and static routing to unique resources, suitable locations are shown. Note that the PowerPC peripherals must be placed on the FPGA and require enough BRAM to support the 64KB of internal memory. In the VC2V2P30, this is 3 BRAM columns. Considering the BRAM requirements and the large number of I/O block connections on the left side of the FPGA, the most logical placement of the PowerPC Peripherals is shown as rectangle surrounded by a dashed line.

NCD files created for the partial bitstream by the partial reconfiguration tools do not pass the PR_verify design stage which checks to make sure that resources used by the static portions of the design are not used by the reconfigurable modules. PR_verify reports that routing for a counter signal used by module that drives the LED display and a global ground signal use "illegal arcs." Defining the boundary of reconfigurable module and prohibiting static routing within reconfigurable areas should have prevented these routes from being placed in the reconfigurable modules. The PR_verify phase of the partial bitstream generation process identifies the illegal use of resources exists and halts the bitstream creation process.

Without valid bitstreams for relocatable modules which do not include any static routing in the reconfigurable areas, internal relocation and reprogramming cannot be demonstrated. To verify that this problem was not fixed in the 8.2 partial reconfiguration toolchain, all components except the PowerPC and peripherals were resynthesized using the ISE 8.2 SP1. The same errors occurred using the 8.2 partial reconfiguration toolchain.

4.7 Relocatable Module Support in 8.2 Partial Reconfiguration Toolchain

Although the ISE 8.2 partial reconfiguration toolchain was not used as the primary toolchain for evaluating relocatable of modules for the Virtex-II Pro, it has additional support for relocatable modules. Although it still in development, Xilinx is adding additional constraints that can be used with the partial reconfiguration design flow to specify that a reconfigurable region is intended to be relocatable [Blo06]. The area group for a reconfigurable module can be specified as being relocatable to other area groups.

Assuming pblock_M1, pblock_M2, pblock_M3 and pblock_M4 has been declared as reconfigurable areas defined on frame boundaries and have the same size and interface, relocation can be specified in the UCF as follows:

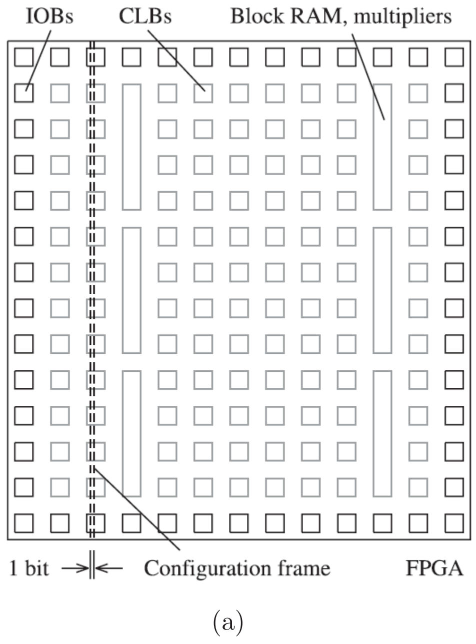
```
AREA_GROUP "pblock_M1" RELOCATABLE=pblock_M2
AREA_GROUP "pblock_M1" RELOCATABLE=pblock_M3

AREA_GROUP "pblock_M1" RELOCATABLE=pblock_M4
```

For the Virtex-II Pro defining an area on frame boundary means the reconfigurable area spans the full height of the FPGA. Additionally, the area groups definition in the UCF must include all I/O blocks included in the reconfigurable areas. When a module is declared as relocatable, the partial reconfiguration tools ensure that the resources used to implement the logic and routing implemented in its original location are available in all of the locations it can be relocated to. Static routing in the relocatable areas is automatically prohibited and no changes to the partial reconfiguration override are needed. In applications where timing is critical, variations of the RELOCATABLE constraint can be used to make sure that resources that effect timing are considered.

Ideally, this constraint could allow a reconfigurable module to utilize the common resources above and below the PowerPC, but on the Virtex-II Pro this is not the case. Attempts to implement a relocatable module which contains any portion of the PowerPC yields an error stating that the PowerPC is in a relocatable module but is not part of the relocatable module. Currently, the only work around is to make sure the relocatable regions do not overlap with the PowerPC [Mas07].

4.8 Relocatable Modules in the Virtex-4

Unlike the reconfiguration frames on the Virtex-II Pro which span the full column of the FPGA, the frames on the Virtex-4 are tiled within the clock regions of the FPGA. Configuration frames in the Virtex-4 are 1-bit wide portions of a CLB column that spans 16 CLBs high. Just as in the Virtex-II, a column in the Virtex-4 is made up of many configuration frames. Figure 4.12 shows the basic configuration architecture for Virtex-II and Virtex-4 devices. A graphical representation of the portion of the FPGA programmed by each configuration frame in each type of device is shown. Also note that the I/O blocks are placed in columns through the FPGA fabric



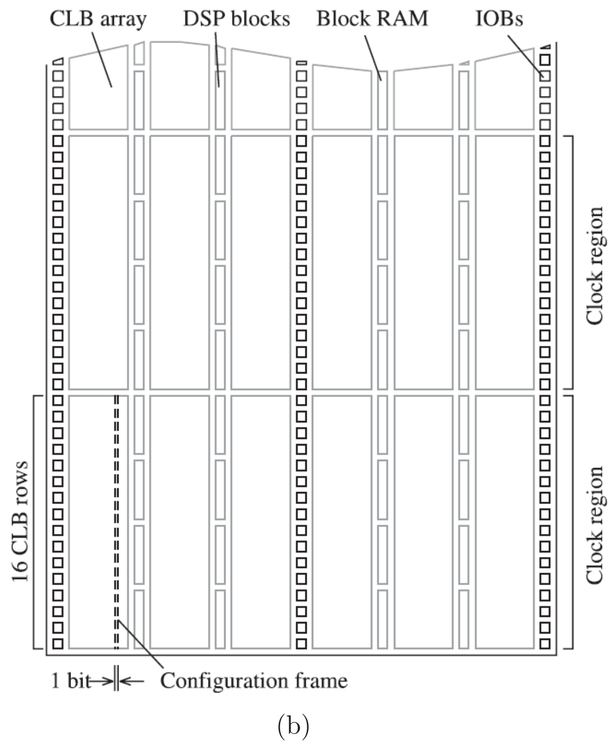


Figure 4.12: Layout of the Virtex-II and Virtex-4. The configuration frame in the Virtex-4 spans the full height of a clock region, spanning only 16 CLB rows instead of the entire height of the column as Virtex-II devices [SBB+06].

and not around the perimeter of the FPGA. The Virtex-4 uses "glitchless" logic and unlike the Virtex-II this includes the LUT RAM and SRL16 logic [SBB+06].

The architecture of the Virtex-4 eliminates many of the problems encountered in implementing relocatable modules on the Virtex-II Pro. Since the configuration frames no longer span the entire height of the device, static routing can go around relocatable modules. This also allows reconfigurable modules to be placed below and above the PowerPCs. Additionally, since I/O block are are not programmed with CLB configuration frames, reconfigurable modules can be moved without reprogramming the I/O blocks.

4.8.1 Drawbacks of the Virtex-4. Bitstream translation for the Virtex-II Pro is possible because the frame addressing scheme is published [Xil05b]. At this time, the frame addressing scheme of the Virtex-4 has not been published. Presumably, the addressing structure is similar to that of the Virtex II-Pro with considerations for the architectural changes. Relocatable modules in the Virtex-4 should be able to be translated vertically or horizontally. Relocation at run-time, in which "all of the frame bits of the module bitstream are shifted by 16 CLBs rows", has been demonstrated using the Virtex-4 [SBB$^+$06] but no details on what was done to shift the frames are included.

4.9 Comparison with REPLICA2Pro

REPLICA2Pro demonstrates that a module can be relocated using hardware MJA translation, but does not apply the technique to fault tolerance. REPLICA2Pro translates bitstreams by changing the MJA in the partial bitstream in the same way the BTP performs bitstream relocation. The primary difference in implementation is that the changing of the bitstream is performed in hardware by REPLICA2Pro instead of software. Although implementing the BTP with a PowerPC and peripherals requires more resources than the REPLICA2Pro, the PowerPC can be used for other functions.

Hardware MJA translation allows REPLICA2Pro to perform bitstream location during the regular internal reconfiguration process. The BTP must translate the entire bitstream before it can be sent to the HWICAP. In the TMR configuration, the system is only able to tolerate one fault at a time, and although the system can continue to produce a correct result, it is operating without redundancy. In each of the TMR configurations developed, the location of the next replacement module is independent of which module becomes faulty. To minimize the amount of time before redundancy is restored, the bitstreams can be translated to the location that will be used for the next spare before they are needed. If "pre-translated" bitstreams are used the reconfiguration time for Replica2Pro and the BTP would be identical.

REPLICA2Pro avoids many of the problems encountered in this research by using Virtex-II modules which are specifically design to connect all important signals on the right side of the FPGA eliminating the use of I/O blocks on the top and bottom of the FPGA that would be reprogrammed by relocating the module. To allow relocation, REPLICA2Pro uses custom designed tools outside of the Xilinx partial reconfiguration toolchain. REPLICA2Pro uses a bus generation system to create a fixed horizontal communication infrastructure that takes the place of bus macros. REPLICA2Pro uses Xilinx ISE 6.3 and generates the partial bitstreams using the PartialMask Bitgen feature. The PR_design and PR_verify commands used to generated bitstreams in the 8.1 and 8.2 toolchains are not available for ISE 6.3.

4.10 Summary

This chapter presents the results of implementing each of the TMR designs on the XUPV2P development board and using the BTP to relocate the bitstreams. The interconnect module and direct connect designs work as expected when unique resources, such as pin connections, are avoided. The LUT-based routing design could not be tested using the 8.1 partial reconfiguration toolchain. The BTP was implemented on the PowerPC and proper operation of the the BTP was verified by comparing the resulting bitstreams with those generated using a standalone PC. Unfortunately, demonstrating an automatic fault recovery system was not possible due to the limitations of the partial reconfiguration toolchain. The architectural features that limit the use of reconfigurable modules were identified and it was determined that the Virtex-4 resolves many of these problems. Finally, the BTP is compared with a hardware implementation that relocated modules using a similar method. The next chapter draws conclusions from these results and presents suggestions for future work.

V. Conclusions

5.1 Introduction

This chapter presents a summary of the problem, the conclusions based on the implementation in Chapter IV and makes suggestions for future research.

5.2 Problem Summary

Partial reconfiguration has been shown to be an effective way to implement fault tolerance in FPGAs. Reprogramming an FPGA to repair a fault requires that a partial bitstream to implemented a replacement module be available at the time of the fault. Since partial bitstreams target specific locations on an FPGA, most previous fault recovery systems pre-generated and stored all of the partial bitstreams needed to implemented a replacement module in each possible location. Upon relocation the module must be connected to the user circuit.

5.3 Conclusion of Research

The goal of this study is to develop an efficient fault recovery system that allows a user circuit to continue to operate through a fault without the need to store individual bitstreams. Due to limitations in the partial reconfiguration tools used to generate the partial bitstreams, and the placement and utilization of resources on the target board, although the basic components of such a system are demonstrated, a functional fault recovery system is not demonstrated.

Three TMR configurations are developed and tested that allow the user circuit to remain operational through a fault and during reconfiguration. The configurations provide passive redundancy and the routing techniques used allow faulty modules to be replaced using translated partial bitstreams. The interconnect module design provides a convenient way to change which result bus each functional module connects to, but the area required to implement the interconnect modules does not justify this benefit. Adding some of the functionality to the interconnect module is a way to reduce the amount of resources left unused in the interconnect module, but the viability of this approach is application dependent.

The direct connect method is straightforward and requires no dedicated space for routing. Although bitstream relocation reduces the number of partial bitstreams needed, the configuration still requires three partial bitstreams to implement the entire functional module and one partial bitstream to program spare modules. Testing of the direct connect design demonstrates that if I/O blocks and unique resources are avoided, modules can be relocated to any reconfigurable location.

The LUT-based design has the potential of being the most efficient method for altering the routing in a reconfigurable module since small bitstreams can be used to change the routing of a standard functional module. Although the principles behind this method were demonstrated, a reconfigurable design which dynamically changes LUT output could not be implemented due to limitations in the partial reconfiguration tool chain which did not incorporate lower level constraints placement.

The BTP successfully translates partial bitstreams to relocate a module by changing the frame address and CRC values in the partial bitstream, eliminating the need to store partial bitstreams for multiple modules that perform the same function but target different locations on the FPGA.

The hypothesis of this study is that the architecture of the Virtex-II Pro and Xilinx partial reconfiguration toolchain allow for the development of a bitstream relocation system which performs bitstream manipulation in software on an embedded microprocessor to relocate partial bitstreams on the FPGA. It was shown that although it is theoretically possible, a usable system cannot be implemented on the XUPV2P development board using the ISE 8.1 partial reconfiguration toolchain. Consolidating many of the steps required for partial reconfiguration into the PR_assemble and PR_verify, Xilinx streamlined the process for the user but reduced the flexibility of the tools. To implement more advanced designs, a higher level of control is needed.

Furthermore, the unique resources on the FPGA must be taken into account when defining relocatable modules to make sure that they can be relocated without disrupting the system. The resources used by the reconfigurable module at its initial location must be available in all locations the module will be relocated to, and static routing must be avoided. Additionally, since configuration data for the I/O blocks are included in each frame, the XUPV2P is a poor target platform for relocating partial bitstreams when using an embedded microprocessor.

5.4 Significance of Research

This research develops a more efficient method for implementing a fault tolerance system using software bitstream translation. Although higher levels of redundancy can be achieved without the need for partial reconfiguration by using N modular redundancy, instantiating the redundant modules before they are needed increases the power consumption of the FPGA. Using TMR with replaceable modules, only three functional modules are instantiated on the FPGA at a time reducing power required. Such a technique could be called "just in time redundancy" since new modules are placed on the FPGA only when they are needed to restore redundancy.

In addition to developing three TMR-based modular designs which take advantage of bitstream translation, the requirements for ensuring that a module can be translated are clearly defined. The resources used in the original location must be available in all of the potential destinations, no static routing can be allowed in the reconfigurable area and the module must not prevent static logic from connecting to unique resources.

5.5 Recommendations for Future Research

A method for relocation modules on the Virtex-II Pro was demonstrated in this study but the target XUPV2P development board did not provide enough usable locations for relocatable modules to properly demonstrate the automatic fault recovery system. Larger versions of Virtex-II and Virtex-II Pro FPGAs have the same architectural features which restrict the placement of reconfigurable modules, such as the PowerPCs on the Virtex-II Pro, but since the FPGAs are larger the restrictions affect a smaller portion of the FPGA. In larger FPGAs, a greater percentage of the FPGA could be used to implement the reconfigurable design.

An alternative to storing the partial bitstreams in memory is to retrieve the configuration data for a module from configuration memory. The retrieved configuration data could be translated and used to create a partial bitstream to relocate the module. This eliminates the need to store partial bitstreams but assumes that the configuration memory has not been corrupted by a SEU and is not corrupting the process reading the configuration data.

With relocatable modules, an alternative approach to using TMR is double modular redundancy. If the system can be stopped when a fault is detected, two modules can be used instead of three. Faults can be detected by monitoring the output of the modules to make sure they match. When a fault occurs the outputs will no longer match and the system halted to allow for recovery. A new module can be created to determine which of the two modules is faulty. Once the faulty module has been identified it can be removed from the system to save power.

The Virtex-4 is a more suitable platform for modular reconfiguration. Many of the limitations imposed by the architecture of the Virtex-II (Pro) are not in the Virtex-4. The fault tolerant designs developed should be adapted and evaluated for use on the Virtex-4. The ability to specify a module as RELOCATABLE in the 8.2 toolchain will greatly reduce the complexity of defining relocatable reconfigurable modules and provide the ability to address timing issues associated with relocation.

Appendix A. Using the PowerPC for Partial Reconfiguration

This appendix describes how to create a PowerPC design using Xilinx Platform Studio that is capable performing partial reconfiguration. This design is based on an Xilinx University Program workshop given by Xilinx on 1 September 2006 in Madrid, Spain.

A.1 Creating the EDK Project

A design using the PowerPC can be built quickly using the Base System Builder (BSB). Before starting the design the board definition package must in installed on the computer. The board definition package for the XUPV2P can be download from `https://www.xilinx.com/univ/xupv2p.html` and can be placed at any location.

Create and New Project

- Select Base System Builder wizard
- Enter the path and filename of the new project
- Check use repository paths and specify the path to the **lib** directory of the board definition
- Click OK
- Verify “I would like to create a new design” is selected and click OK

Select Board The target developed board can be selected. Note that if the repository path correctly points to the board definition only the correct board vendor, name and revision will be available in the drop down menus.

- Select Board vendor: Xilinx
- Select Board name: XUP Virtex-II Pro Development System
- Select Board revision: C

Select Processor The FPGA selection drop down menus should be grayed out. The next series of screens allow for customizations to the PowerPC and peripherals. The ICAP works at the same speed at the OBP, so the bus clock frequency and processor frequency do not need to be adjusted.

- Select PowerPC

Configure PowerPC Select the following:

- Set Processor clock frequency to 100MHz
- Set Bus clock frequency 100MHz
- Select FPGA JTAG for the Debug I/F
- Check the enable box for cache setup
- Enable cache setup
- Select None for both Data and Instruction On-chip memory

Configure I/O devices as follows. All other I/O devices should not be included (uncheck):

Universal Asynchronous Reciever/Transmitter RS232_Uart1

- Peripheral: OPB UARTLITE
- Baudrate: 9600
- Data bits: 8
- Parity: None
- Use Interrupt should **not be** checked

SystemACE for Compact Flash SysACE_CompactFlash

- Peripheral: OPB SYSACE

- Use Interrupt should **not be** checked

Extended External Memory DDR_256MB_32MX64_rank1_row13_col10_cl2_5

- Peripheral: PLB DDR
- Use Interrupt should **not be** checked

Note that is could be different if a difference size memory module is used.

BRAM controller pbl_bram_if_cntlr_1

- Peripheral: PLB BRAM IF CNTLR
- Memory size: 64KB

Cache Setup Since all instructions will be stored in the BRAM, instruction cache is not needed for the external memory.

- DDR_256MB_32MX64_rank1_row13_col10_cl2_5: Check DCache
- pbl_bram_if_cntlr_1: Check both ICache and DCache

Software Setup

- STDIN: RS232_Uart_1
- STDOUT: RS232_Uart_1
- Check the memory test checkbox
- Uncheck the Peripheral selftest

Memory Test

- Instruction: pbl_bram_if_cntlr_1
- Data: pbl_bram_if_cntlr_1
- Stack/Heap: pbl_bram_if_cntlr_1

A.2 Adding Software, Exporting, and Integration

Once the base system has been generated, two additional peripherals can be added. To allow for partial reconfiguration the obp_hwicap must be added to the design. The opb_timer is used to measure the time it takes for the PowerPC to alter the bitstream and reprogram the FPGA and can be accessed from the PowerPC using Xilinx drivers. To add these components, find them in the IP catalog, right click on them and select add IP. Both of these devices connect to the OPB. Once they are visible in the System Assembly view, click on the hollow green circles on the OPB bus to connect each device to the bus. To assign addresses to the HW_ICAP and timer, select the "Address" filter and click on the "Generate Addresses" button.

The design can be tested using the memory test program. Next, software is added using the applications tab and support for the FAT16 file system on the Compact Flash is added by selecting xilfatfs in the Software Platform Setting Menu.

To prepare the design to be exported into ISE, the clock buffer for the main system added by EDK must be removed. Right click on dcm_0, select configure IP, select buffers and choose "False" for "Insert a BUFG for CLK0". This buffer must be added at the top level so that it can also be used as the clock of the reconfigurable design. The project can now be exported into ISE. First change the setting on the Hierarchy and Flow Tab in the Project Options menu. Check both the "Processor Design is a sub-module" and the "Use Project Navigator Flow" boxes. Then select "Export Project to ISE" from the Project menu.

Exporting the EDK project creates a new ISE project with a top level file system_stub.vhd which instantiates the PowerPC system and defines the proper pin connections. A system.ucf file is also created with the pin assignment. The system must be synthesized by running XST. Using the instantiation template for the PowerPC design from system_stub.vhd, a top level design which includes both the microprocessor and the reconfigurable areas can be created. The clock buffer removed from the EDK project must be replaced with a clock buffer at the top level.

Appendix B. Bitstream Translation Programs

This appendix contains the functions used to translate to the PowerPC. Unlike the PC version, module translation distance is specified in CLBs and device specific attributes such as the number of CLB columns in the device are hard coded for efficiency. The translate() and supporting functions changes the frame address and CRC values for a bitstream in memory given a pointer to the start of the bitstream in memory, the size of the bitstream, and the distance in CLBs to translate the module. The following definitions can be used for debugging and customization:

```
#define verbose     0
#define basic_info  0
#define crc_debug   0
#define bypass_crc  0
#define show_crc    0
#define CLB_width   6
```

The translate function starts on page 97 and is preceded by the functions it calls.

```
/*********************************************************************
 The update_BCC function updates the bcc value which is used to calculate
 the CRC. The BCC/CRC is determined by both the word being writen and the
 address being written to. This function returns the bcc value.
 get_CRC(bcc) returns the CRC value based on bcc.
*********************************************************************/
unsigned int update_BCC(unsigned int bcc, unsigned int current_word,          update_BCC
      int current_reg ){
  unsigned long sw36_32, sw31_0;   // used variable from
  unsigned int sw, x16, x15, x2;   // XAPP151
  int i, j, addr, word;
  addr = current_reg;
  word = current_word;
  sw36_32 = addr;
  sw31_0 = word;
  for (i=0; i<37; i++){ // iterate over the 37 bit input to CRC function
  if (i<32){ j=i;       sw = ((sw31_0 >> j) & 1); } // if i<32 used sw31_0
  else {j=i−32; sw = ((sw36_32 >> j) & 1);} // if i<=32 use sw36_32
  x16 = (bcc >> 15)^(sw);                         // bcc[15] XOR sw[j]
  x15 = (((bcc >> 14)^(x16)) & 1); // bcc[15] XOR x16
  x2 = (((bcc >> 1 )^(x16)) & 1);   // bcc[1] XOR x16
  bcc = ((x15 << 15) | ((bcc & 0x3FFC) << 1)
             | (x2 << 2) | ((bcc & 1) << 1) | (x16));
  }
  if (show_crc){
  xil_printf(" (Data: %4X Reg: %4X BCC:%4X) \n\r ", word, addr, bcc);
  }
  return bcc;
}// end of update_BCC update
/************************************************************************
 The getCRC function take the value of BCC and reverses the bits to get
 the current CRC value. The code to reverse the bits in CRC crc[0..15]=
 bcc[15..0] found at http://graphics.stanford.edu/~seander/bithacks.html
************************************************************************/
unsigned int get_CRC(unsigned int bcc){                                    get_CRC

      int i = 0;
      unsigned int crc;
      unsigned int rem_me = bcc;
      crc = bcc << 1;
      bcc >>= 1;
      for (i =16 − 2; i; i−−)
      {
      crc |= bcc & 1;
      crc <<= 1;
      bcc >>= 1;
      }
      crc |= bcc;
      crc = crc & 0xFFFF;
      bcc = rem_me;

      if (show_crc){
      xil_printf(" BCC:%4X CRC:%4X ", bcc, crc);
      }
      return crc;
}// end of get_CRC
/*********************************************************************
 The change_address function changes the location of the modules
 contained in the partial bitstream by changing the major address of for
 the columns. It recognizes the block address of the frame address and
 makes the appropriate changes using the global variables describing
 the architecture of modular configuration. Frame Address Composition:

                    BA      MJA      MNA    Byte Number
   31-27    26-25    24-17     16-9          8-0
*********************************************************************/
unsigned int change_address(unsigned int old_address, int dist_in_mods)          change_address
{       unsigned int new_address;
      unsigned int new_mja;
      unsigned int new_mja_confirm; // used to verify shift worked properly
      unsigned int block_type = ((old_address & 0x06000000) >> 25);
      unsigned int mja = ((old_address & 0x01FE0000) >> 17);

      /*****************************************************************
 Define the architecture of the modular design here
*****************************************************************/
int n_clb =   46;     // number of clb columns (device dependent)
int m_ram =  16;      // number of ram columns (device dependent)
int mod_width = 4;    // width in CLB colums of functional modules
int inter_width = 2; // width in CLB colums of interconnect modules
int CLBs_between_RAM = 6; // number of CLBs between RAM columns
/*****************************************************************/
if (block_type==0){ // for CLB columns
        if (verbose){ xil_printf("CLB Column ");}
        new_mja = mja + (dist_in_mods * (mod_width + inter_width));
        // calculate the new major address
        new_mja = new_mja << 17;
        // shift the major address into position
        new_address = ((old_address & 0xFE01FFFF) + new_mja);
        // put the unchanged bits back around the the new MJA
}
else if(block_type==1){ // for BRAM Column
        if (verbose){ xil_printf("BRAM Column ");}
        new_mja = mja + (dist_in_mods * (mod_width + inter_width)
                /CLBs_between_RAM);
        // calculate the new major address
        new_mja = new_mja << 17;
        // shift the major address into position
        new_address = ((old_address & 0xFE01FFFF) + new_mja);
        // put the unchanged bits back around the the new MJA
}
else if(block_type==2){ // BRAM Column Interconnect
        if (verbose){ xil_printf("BRAM Interconnect Column "); }
        new_mja = mja + (dist_in_mods * (mod_width + inter_width)
                /CLBs_between_RAM);
```

```
            // calculate the new major address
            new_mja = new_mja << 17;
            // shift the major address into position
            new_address = ((old_address & 0xFE01FFFF) + new_mja);
            // put the unchanged bits back around the the new MJA
        }
        else{
        xil_printf("ERROR: Unknown Column Type\n\r");
        // shouldn't happen
        }
        new_mja_confirm = ((new_address & 0x01FE0000) >> 17);
        //extracts MJA to check
        if (verbose){
        xil_printf("Frame Address: BA:%d MJA:%d => MJA:%d\n\r",
            block_type, mja, new_mja_confirm);
        }
        return new_address;
} //end of change address()

/*********************************************************************
 The decode_command functions allows for special actions for each of the
 possible commands that are sent to the command register. Most important
 is clearing the CRC register when the CRC is written to and when the
 pulse GCAPTURE signal is detected signifing the end of the bitstream.
*********************************************************************/
unsigned int decode_command(unsigned int command_code, unsigned int bcc)
{
char command_name[30]= "no command";
bcc = update_BCC(bcc, command_code, 4); // command_reg = 4
switch(command_code) // determine type of instruction
{
```

```
case (1) : { strcpy(command_name, "Write Configuration Data"); break;}
case (2) : { strcpy(command_name, "Multiple Frame Write Register"); break;}
case (3) : { strcpy(command_name, "Last Frame"); break;}
case (4) : { strcpy(command_name, "Read Configuration Data"); break;}
case (5) : { strcpy(command_name, "Begin Startup Sequence"); break;}
case (6) : { strcpy(command_name, "Reset Capture"); break;}
case (7) : { strcpy(command_name, "Reset CRC");
   bcc = 0;
      if (crc_debug){ xil_printf("*CRC RESET* ");} break;}
case (8) : { strcpy(command_name, "Assert GHIGH_B Signal"); break;}
case (9) : { strcpy(command_name, "Switch CCLK Frequency"); break;}
case (10): { strcpy(command_name, "Pulse the GRESTORE Signal"); break;}
case (11): { strcpy(command_name, "Begin Shutdown Sequence"); break;}
case (12): { strcpy(command_name, "Pulse GCAPTURE Signal"); break;}
case (13): {
      if (verbose){
      xil_printf("Command Code : Reset DALIGN Signal \n\r");
      } // this signal indicates the end of the bitstream
      if(basic_info){ xil_printf("End of bitstream found.\n\r");}
      break;}
default : { strcpy(command_name, "Warning: UNKNOWN COMMAND \n\r");
                  xil_printf("%s \n\r", command_name);
                  break;}
      }
if (verbose && (command_code != 13)){
xil_printf("Command Code : %s Returned BCC: %X \n\r",
               command_name, bcc);
}
      return bcc;
} // end of command ID
```

```
/*********************************************************************
 The id_device identifes the device type by getting the next word after
 a Type 1 instruction to write to the device id register. The word is
 decoded to the device name.
*********************************************************************/
unsigned int id_device(unsigned int device_id_code, unsigned int bcc)
{
      char device_name[15];
      bcc = update_BCC(bcc, device_id_code, 14); // device register = 14
      //write_word(device_id_code); // never change device code
      switch(device_id_code)    // determine name of the device
      {
      case (19030163) : { strcpy(device_name, "XC2VP2"); break;}
      case (19128467) : { strcpy(device_name, "XC2VP4"); break;}
      case (19177619) : { strcpy(device_name, "XC2VP7"); break;}
      case (19292307) : { strcpy(device_name, "XC2VP20"); break;}
      case (25583763) : { strcpy(device_name, "XC2VPX20"); break;}
      case (19390611) : { strcpy(device_name, "XC2VP30"); break;}
      case (19472531) : { strcpy(device_name, "XC2VP40"); break;}
      case (19521683) : { strcpy(device_name, "XC2VP50"); break;}
      case (19636371) : { strcpy(device_name, "XC2VP70"); break;}
      case (25927827) : { strcpy(device_name, "XC2VPX70"); break;}
      case (19751059) : { strcpy(device_name, "XC2VP100"); break;}
      default : { strcpy(device_name, "UNKNOWN DEVICE"); break;}
      }
      if(basic_info)
      {
      xil_printf("Bitstream Target Device: %s\n\r", device_name);
      }
      return bcc;
}
```

```
/************************************************************************
 The translate function determines what type of instructions each word
 is and calls all related functions based on the type of instruction.
************************************************************************/
void translate(Xuint32 *bsPtr, Xuint32 bsSize, Xuint32 distance){
int i = 0; // used for master pointer
int j = 0; // used in local loops
int register_num; // holds the register being accesses
int bcc = 0; // calculated BCC
int bsCRC = 0; // CRC in bitstream
int new_address; // the new address after translation
int word_count; // the number of words listed in the instruction
int auto_CRC; // the auto_CRC from the original bitstream
char register_name[40]="no reg"; // decoded register name
// find the synchronization sequence
while (i < bsSize){
	if (bsPtr[i]==0xAA995566){
	if(basic_info){
	xil_printf("Synchronization Sequence Found\n\r");
	xil_printf("Translating Bitstream %d modules\n\r", distance);
	}
	i++;
	break;
	}
	i++;
	if (i >= bsSize){
	xil_printf("Synchronization Failed");
	exit(1);
	}
}
while (i < bsSize){
```

```
// Use the bitstream size to determine when to stop
if(verbose){ xil_printf("0x%x | ", bsPtr[i]);}
// displays the HEX for each command (not skipped words or addresses)
if ((bsPtr[i] & 0xF8000000) == 0x28000000) // type 1 read op
{//start type 1 read
word_count = bsPtr[i] & 0x000007FF;
  for (j=0; j<word_count; j++){ // skip words
    i++; // skip words in the bitstream (no CRC for read)
  }
  if (verbose){
  xil_printf("Type 1 read operation: reading %i words\n\r", word_count);
  }
}// end type 1 read
else if ((bsPtr[i] & 0xF8000000) == 0x48000000)
{//Type 2 read operation
  word_count = bsPtr[i] & 0x07FFFFFF; // extract word count
  for (j=0; j<word_count; j++){ // skip words
  i++; // skip words in the bitstream (no CRC for read)
  }
  if (verbose){
  xil_printf("Type 2 read operation: reading %i words\n\r", word_count);
  }
}// end type 2 read operation
else if ((bsPtr[i] == 0x20000000))
{// Type 1 No OP
  if (verbose){ xil_printf("Type 1 NOOP word 0\n\r"); }
        i++;
}
else if ((bsPtr[i] & 0xF0000000) == 0x30000000) //type 1 write
{
register_num = ((bsPtr[i] & 0x07FFE000) >> 13);
```

```
// extract the register from the instruction
word_count = bsPtr[i] & 0x000007FF; // Determine word count
i++;
switch(register_num) // Determine name of register
// this greatly helps in debugging allowing the bitstream to be readable
{
case (0) : { strcpy(register_name, "CRC Register"); break;}
case (1) : { strcpy(register_name, "Frame Register Address"); break;}
case (2) : { strcpy(register_name, "Frame Data Input Register"); break;}
case (3) : { strcpy(register_name, "Frame Data Output Register"); break;}
case (4) : { strcpy(register_name, "Command Register"); break;}
case (5) : { strcpy(register_name, "Control Register"); break;}
case (6) : { strcpy(register_name, "Masking Register for CTL"); break;}
case (7) : { strcpy(register_name, "Status Register"); break;}
case (8) : { strcpy(register_name, "Legacy Output Register"); break;}
case (9) : { strcpy(register_name, "Configuration Option Register"); break;}
case (10) : { strcpy(register_name, "Multiple Frame Write Register"); break;}
case (11) : { strcpy(register_name, "Frame Length Register"); break;}
case (12) : { strcpy(register_name, "Initial Key Address Register"); break;}
case (13) : { strcpy(register_name, "Initial CBC Value Register"); break;}
case (14) : { strcpy(register_name, "Device ID Register"); break;}
}
if (verbose) { xil_printf("Type 1: Write %d word(s) to %s
      register\n\r", word_count, register_name); }
            switch(register_num) {
            // Decode register and perform required actions
case (0) : { // CRC Register
      if (crc_debug){ xil_printf("Calculated Explicit CRC: %X ", get_CRC(bcc));}
      bsCRC = bsPtr[i++]; // get value of CRC in bitstream for comparison
      if (bypass_crc){
            // do nothing
```

```
        }
    else {
    bsPtr[i-1] = get_CRC(bcc); // write over old CRC value
    };
    if (crc_debug){ xil_printf("Bitstream Explicit CRC: %X\n\r", bsCRC);
    // identified this as an explicit bitstream CRC write
    }
    break;}
case (1) :
    { // Frame Register Address
    new_address = change_address(bsPtr[i], distance);
    // calculated the new address based on old and translation distance
    bcc = update_BCC(bcc, new_address, register_num);
    // update BCC based on the new address
    bsPtr[i] = new_address; // write new address to memory
    i++; // points to the next command
    break;}
case (2) : { // Frame Data Input Register
    // pass through each of the configuration words
    if (word_count > 0){// skip words with CRC update
     for (j=0; j<word_count; j++){
     bcc = update_BCC(bcc, bsPtr[i], register_num);
     i++; // skip words in the bitstream
     } // end of for
     if (crc_debug){ xil_printf("Calculated CRC: %X ", get_CRC(bcc));
     // prints the current value of crc
     }
     auto_CRC = bsPtr[i]; // get the bitstream's auto_CRC for comparison
                                          // during testing
     if (bypass_crc){
          //do nothing
     } // end of if
     else {
     bsPtr[i++] = get_CRC(bcc); // writes new CRC value
     } // end of else
     if (crc_debug){xil_printf("Auto CRC: %X\n\r", auto_CRC, get_CRC(bcc));}
     // Displays the crc value from the bitstream
     bcc = 0; // clears the bcc register (therefore crc)
    } // end of if word_count >0
    break;}
case (3) : { // Frame Data Output Register                                    case
    if (word_count > 0){// skip words with CRC update
     for (j=0; j<word_count; j++){
     bcc = update_BCC(bcc, bsPtr[i++], register_num);
     // skip words in the bitstream and update CRC
     }
    } // end of skip words with CRC update
    break;}
case (4) : { // Command Register                                              case
    bcc = decode_command(bsPtr[i++], bcc);
    break;  }
case (5) : { // Control Register                                              case
    if (word_count > 0){// skip words with CRC update
     for (j=0; j<word_count; j++){
     bcc = update_BCC(bcc, bsPtr[i++], register_num);
     // skip words in the bitstream and update CRC
     }
    } // end of skip words with CRC update
    break;}
case (6) : { // Masking Register for CTL                                      case
    if (word_count > 0){// skip words with CRC update
     for (j=0; j<word_count; j++){
```

```
      bcc = update_BCC(bcc, bsPtr[i++], register_num);
      // skip words in the bitstream and update CRC
      }
    } // end of skip words with CRC update
    break;}
case (7) : { // Status Register                                   case
    if (word_count > 0){// skip words with CRC update
      for (j=0; j<word_count; j++){
      bcc = update_BCC(bcc, bsPtr[i++], register_num);
      // skip words in the bitstream and update CRC
      }
    } // skip words with CRC update
    break;}
case (8) : { // Legacy Output Register                            case
    if (word_count > 0){// skip words with NO CRC update
      for (j=0; j<word_count; j++){
      i++; // skip words in the bitstream
      // legacy output does not update CRC
      }
    } // skip words with NO CRC update
    break;}
case (9) : { // Configuration Option Register                     case
    if (word_count > 0){// skip words with CRC update
      for (j=0; j<word_count; j++){
      bcc = update_BCC(bcc, bsPtr[i++], register_num);
      // skip words in the bitstream and update CRC
      }
    } // end of skip words with CRC update
    break;}
case (10) : { // Multiple Frame Write Register                    case
    if (word_count > 0){// skip words with CRC update
```

```
      for (j=0; j<word_count; j++){
      bcc = update_BCC(bcc, bsPtr[i++], register_num);
      // skip words in the bitstream and update CRC
      }
    } // skip words with CRC update
    if (verbose){
    xil_printf("packet data write MFMR word %i\n\r", i);
    }
    break;}
case (11) : { // Frame Length Register                                    case
    if (word_count > 0){// skip words with CRC update
      for (j=0; j<word_count; j++){
      bcc = update_BCC(bcc, bsPtr[i++], register_num);
      // skip words in the bitstream and update CRC
      }
    } // skip words with CRC update
    break;}
case (12) : { // Initial Key Address Register                             case
    if (word_count > 0){// skip words with CRC update
      for (j=0; j<word_count; j++){
      bcc = update_BCC(bcc, bsPtr[i++], register_num);
      // skip words in the bitstream and update CRC
      }
    } // skip words with CRC update
    break;}
case (13) : { // Initial CBC Value Register                               case
    if (word_count > 0){// skip words with CRC update
      for (j=0; j<word_count; j++){
      bcc = update_BCC(bcc, bsPtr[i++], register_num);
      // skip words in the bitstream and update CRC
      }
```

```
        } // skip words with CRC update
        break;}
case (14) : { // Device ID Register                                        case
        bcc = id_device(bsPtr[i++], bcc);
        break;
        }
}
} // end of else if
else if ((bsPtr[i] & 0xE0000000) == 0x40000000)                           if
{ // Type 2 write
  word_count = bsPtr[i++] & 0x07FFFFFF;
  if(verbose){
    xil_printf("Type 2: Write %d word(s) to last register\n\r", word_count);
  }
  for (j=0; j<word_count; j++){ // skip words
          bcc = update_BCC(bcc, bsPtr[i++], register_num);
          // skip words in the bitstream and update CRC
  } // end skip words
         auto_CRC = bsPtr[i++]; // get the auto_CRC from the bitstream
         if (crc_debug){xil_printf("Auto CRC: %X
                 Calculated CRC: %X \n\r", auto_CRC, get_CRC(bcc));}
         // Displays the crc value from the bitstream
         bcc=0; // reset the bcc (and crc)
} //end Type 2 write else if

else { // catch all
xil_printf("WARNING: Unknown Type\n\r");
i++;
}

} // end of while statement that searches for end of file
} // end of translate function
```

Appendix C. ISE 8.1 and PlanAhead Design Flow

The design flow for partial reconfiguration continues to change as Xilinx improves the tools available to create partially reconfigurable designs. The latest documentation is available on Xilinx's Early Access Partial Reconfiguration web site.

C.1 ISE 8.1 Partial Reconfiguration Design Flow

The partial reconfiguration design flow for 8.1 (without using PlanAhead) is found in [Xil06]. The user guide includes the requirements for partial configuration designs including those with EDK components. Instruction on using PlanAhead for partial reconfiguration can also be found at this site.

PlanAhead greatly simplifies the process of creating partial bitstreams. Once the top level and each of the static and reconfigurable modules have been synthesized using XST, only the .ncd, .ucf, and bus macro .nmc files are needed to create the partial reconfiguration design. Once area group constraints and bus macros have been placed using the graphical interface, the built-in design rule checker can be used to verify the partial reconfiguration design rules have been met. To perform the partial reconfiguration design flow, PlanAhead takes the .ncd, .ucf, and .nmc files and copies them to the appropriate directories. Scripts to perform the necessary ISE actions are automatically generated and can be run from within PlanAhead. Although PlanAhead makes constructing a basic partial reconfiguration design easier, the scripts generated by PlanAhead do not extend the design flow to included generation of the ACE file or programming the microprocessor included in a partial reconfiguration design.

For designs with microprocessor or for more flexibility, batch scripts can be tailored to implement the partially reconfigurable design. Figures C.1 and C.2 illustrate the complexity of the partial reconfiguration design.

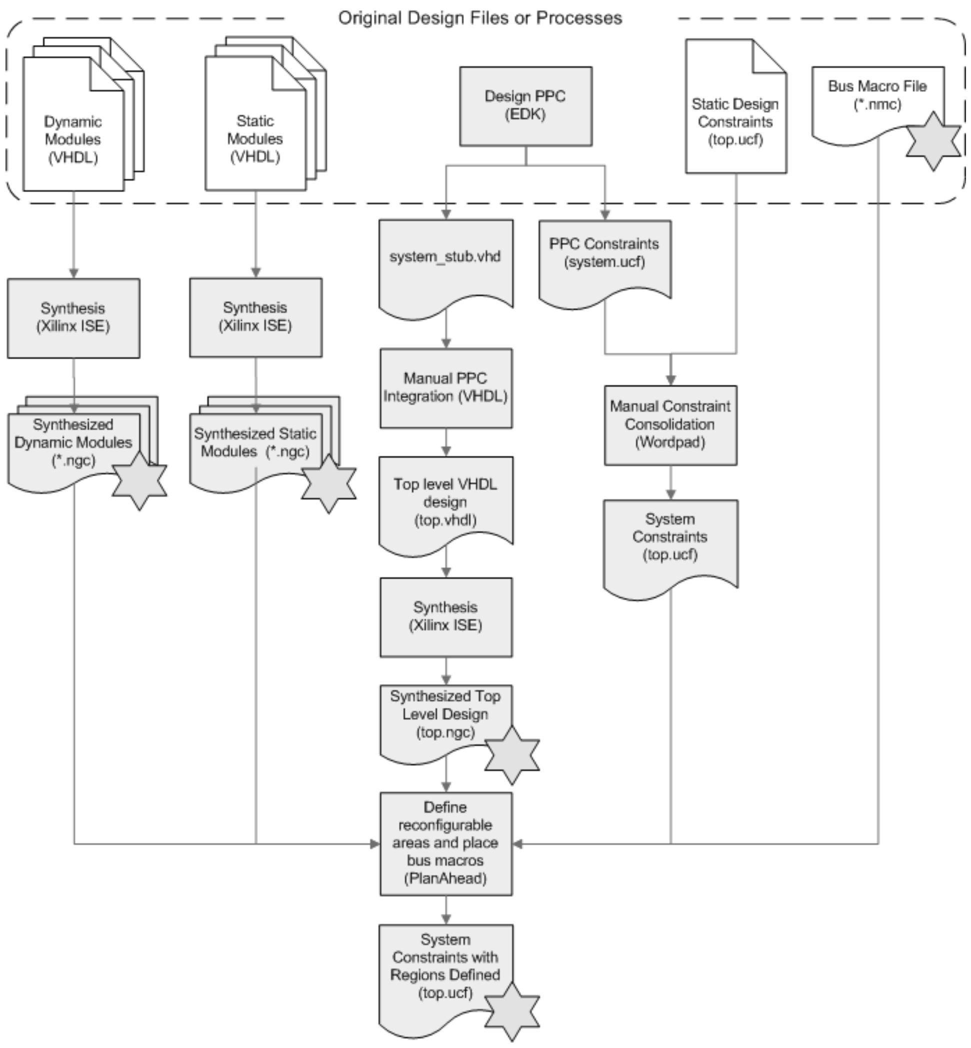

Original Design Files or Processes
Dynamic Modules (VHDL)
Static Modules (VHDL)
Design PPC (EDK)
Static Design Constraints (top.ucf)
Bus Macro File (*.nmc)
system_stub.vhd
PPC Constraints (system.ucf)
Synthesis (Xilinx ISE)
Synthesis (Xilinx ISE)
Manual PPC Integration (VHDL)
Manual Constraint Consolidation (Wordpad)
Synthesized Dynamic Modules (*.ngc)
Synthesized Static Modules (*.ngc)
Top level VHDL design (top.vhdl)
System Constraints (top.ucf)
Synthesis (Xilinx ISE)
Synthesized Top Level Design (top.ngc)
Define reconfigurable areas and place bus macros (PlanAhead)
System Constraints with Regions Defined (top.ucf)

Figure C.1: Partial Reconfiguration Design Flow (1). The design starts as individual VHDL files for each of the static and reconfigurable modules. The top level design (top.vhd) is created by adding top-level logic such as bus macros, connections between static modules, pin connects, and clock buffers to the system_stub.vhd created in EDK. For easy module and bus macro placement, the synthesized modules, system constraint file and bus macros can be loaded into PlanAhead even if PlanAhead is not used to generate the scripts.

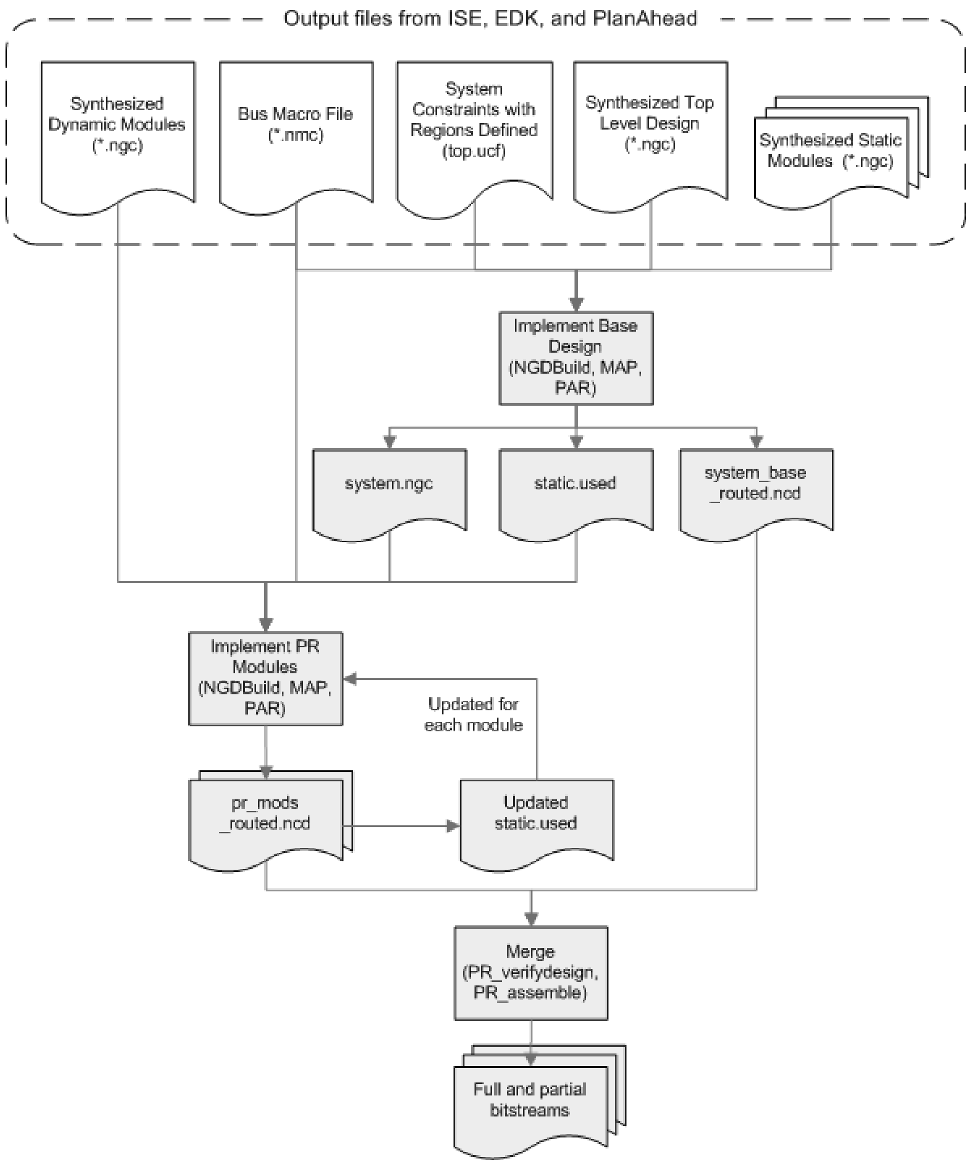

Output files from ISE, EDK, and PlanAhead
Synthesized Dynamic Modules (*.ngc)
Bus Macro File (*.nmc)
System Constraints with Regions Defined (top.ucf)
Synthesized Top Level Design (*.ngc)
Synthesized Static Modules (*.ngc)
Implement Base Design (NGDBuild, MAP, PAR)
system.ngc
static.used
system_base _routed.ncd
Implement PR Modules (NGDBuild, MAP, PAR)
Updated for each module
pr_mods _routed.ncd
Updated static.used
Merge (PR_verifydesign, PR_assemble)
Full and partial bitstreams

Figure C.2: Partial Reconfiguration Design Flow (2). The synthesized top level design and module are combined in this phase to implement the base design and each of the PR modules. The designs are merged together using the PR_verify and PR_assemble functions which first verify that all partial reconfiguration rules are followed then produce the partial, blanking and full bitstreams.

C.2 ISE 8.2 Partial Reconfiguration Toolchain

The Xilinx 8.2 partial reconfiguration tools chain was also experimented with to determine it provided a greater level of support for partial reconfiguration. Although it was not used because of the deprecated flow for exporting EDK designs to ISE, one key changes was discovered.

Stating with the 8.2, the partial reconfiguration toolchain creates compressed bitstreams by default. Data2mem.exe can not be used with a compressed bitstream. To prevent the bitstreams from being compresses the `"-g compress:no"` option must be used with PR_verifydesign and PR_assemble. To use the following script in 8.2, this option must be added.

C.3 Example PR Implementation Script

The following script is based on an Xilinx University Program workshop given by Xilinx on 1 September 2006 in Madrid, Spain. For brevity only two reconfigurable modules are used.

```
# Build top level context
echo -e " \ nStart: 1) Build top level context\n"
cd Top
rm *
cp ../Synth/Top/top.ngc .
cp ../Data/top.ucf .
cp ../Data/*.nmc .
ngdbuild -modular initial -p xc2vp30-7-ff896 top.ngc
cd ..
# build static portion of the design
echo -e "\nStart: 2) Build static portion of
    the design\n"
cd Static
rm *
```

```
cp ../Synth/display_mem/display_mem.ngc .
cp ../Synth/led_driver/led_driver.ngc .
cp ../Synth/static_in/static_in.ngc .
# next 3 lines are required for edk project
cp ../Synth/edk/implementation/*.ngc .
cp ../Synth/edk/implementation/system_stub.bmm .
cp ../Synth/edk/projnav/*.ngc .
cp ../Data/top.ucf .
cp ../Data/*.nmc .
ngdbuild -p xc2vp30-7-ff896 -bm system_stub.bmm
    -modular initial ../Top/top.ngo
map top.ngd
par -w top.ncd top_routed.ncd
cd ..
# Build m1_mod reconfig module
echo -e "\nStart: 4a) Build m1_mod reconfig module\n"
cd ReconfigModules/m1_mod
rm *
cp ../../Synth/m1_mod/function_mod1.ngc .
cp ../../Data/top.ucf .
cp ../../Data/*.nmc .
cp ../../Static/static.used arcs.exclude
ngdbuild -modular module -p xc2vp30-7-ff896
    -active function_mod1 ../../Top/top.ngo
map top.ngd
par -w top.ncd top_routed.ncd
cd ../..
# Build m2_mod reconfig module
echo -e "\nStart: 4b) Build m2_mod reconfig module\n"
cd ReconfigModules/m2_mod
rm *
```

```
cp ../../Synth/m2_mod/function_mod2.ngc .
cp ../../Data/top.ucf .
cp ../../Data/*.nmc .
cp ../../Static/static.used arcs.exclude
ngdbuild -modular module -p xc2vp30-7-ff896
    -active function_mod2 ../../Top/top.ngo
map top.ngd
par -w top.ncd top_routed.ncd
cd ../..
# Merge ncds and generate bitstreams
echo -e "\nStart: 7) Merge ncds and generate bitstreams\n"
cd Merges
rm *
rm -rf PRtmpdir
cp ../Static/top_routed.ncd static.ncd
cp ../ReconfigModules/m1_mod/top_routed.ncd function_mod1_routed.ncd
cp ../ReconfigModules/m2_mod/top_routed.ncd function_mod2_routed.ncd
# next line required for edk design
cp ../Synth/edk/implementation/system_stub.bmm .
PR_verifydesign.bat static.ncd function_mod1_routed.ncd function_mod2_routed.ncd
PR_assemble.bat static.ncd function_mod1_routed.ncd function_mod2_routed.ncd
cd ..
# Create download.bit
cd Merges
echo -e "\nStart: 8) Create download.bit \n"
cp ../Synth/edk/TestApp_Reconfig/executable.elf .
data2mem -bm system_stub_bd.bmm -bt static_full.bit
    -bd executable.elf tag plb_bram_if_cntlr_1_bram -o b download.bit
cd ..
echo -e "\nStart: 9) Copy bitstreams back to EDK project \n"
cd Merges
```

```
cp static_full.bit ../Synth/edk/implementation/system.bit
cp download.bit ../Synth/edk/implementation
cp system_stub_bd.bmm ../Synth/edk/implementation/system_bd.bmm
cd ..
# Create system.ace
echo -e "\nStep 10) - Creating system.ace file \n"
cd Merges
rm ../CF_files/*
cp ../Data/genace.opt .
cp ../Data/genace.tcl .
xmd -tcl ./genace.tcl -opt genace.opt
cp system.ace ../CF_files
cp function_mod1_routed_partial.bit ../CF_files/m1_mod.bit
cp function_mod2_routed_partial.bit ../CF_files/m2_mod.bit
cp pblock_m1_blank.bit ../CF_files/m1_blank.bit
cp pblock_m2_blank.bit ../CF_files/m2_blank.bit
cd ..
echo -e "\nDone!\n"
```

Bibliography

AL81. T. Anderson and P. Lee. *Fault Tolerance Principles and Practice*. Prentice Hall, 1981.

Alp98. C. J. Alpert. The ISPD98 circuit benchmark suite. In *Proceedings of the 1998 international symposium on Physical design*, pages 80–85, 1998.

BJRK$^+$03. Brandon Blodget, Philip James-Roxby, Eric Keller, Scott McMillan, and Prasanna Sundararajan. A Self-reconfiguring Platform. In *Lecture Notes in Computer Science*, volume 2778, pages 565–574, September 2003.

Blo06. B. Blodget. Research Engineer, Xilinx Labs, Logmont, CO. Personal Coorespondance. 11 December 2006.

CCMM04. E. Carvalho, N. Calazans, F. Moraes, and D. Mesquita. Reconfiguration Control for Dynamically Reconfigurable Systems. In *Proceedings of Conference On Design of Circuits and Integrated Systems (DCIS)*, pages 405–410, 2004.

DFR$^+$05. Alberto Donato, Fabrizo Ferrandi, Massimo Redaellii, Marco D. Santambrogio, and Donatella Sciuto. Caronte: a complete methodology for the implementation of a partially dynamically self-reconfigurating systems on FPGA platforms. *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 321–322, 2005.

DP94. S. Durand and C. Piguet. FPGA with Self-Repair Capabilities. *ACM Int Workshop on Field-Programmable Gate Arrays (FPGA94), Berkeley, February*, pages 1–6, 1994.

FHA03. R.J. Fong, S.J. Harper, and P.M. Athanas. A versatile framework for FPGA field updates: an application of partial self-reconfiguration. In *Rapid Systems Prototyping, 2003. Proceedings. 14th IEEE International Workshop on*, pages 117–123, 2003.

GAF05. M. Gericota, G. Alves, and J. Ferreira. Robust Configurable System Design with Built-In Self-Healing. In *Conference on Design of Circuits and Integrated Systems*, 2005.

GLS99. S. Guccione, D. Levi, and P. Sundararajan. JBits: A Java-based interface for reconfigurable computing. *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD).*, 1999.

HL01. E.L. Horta and J.W. Lockwood. *PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays.*

Department of Computer Science, Applied Research Lab, Washington University, Tech Rep. WUSC-01-13 edition, July 2001.

HM01. W-J. Huang and E.J. McCluskey. Column-Based Precompiled Configuration Techniques for FPGA. *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pages 137–146, 2001.

Ive06. J. Ives. Evaluation of a Field Programmable Gate Array Circuit Reconfiguration System. Master's thesis, Air Force Institute of Technology, 2006.

Kha02. Jamil Khatib. Introduction to Programmable Logic Devices. 2002. http://www.geocities.com/jamilkhatib75/fpga/FPGA_intro.html.

KJdlTR05. YE Krasteva, AB Jimeno, E. de la Torre, and T. Riesgo. Straight Method for Reallocation of Complex Cores by Dynamic Reconfiguration in Virtex II FPGAs. *The 16th IEEE International Workshop on Rapid System Prototyping*, pages 77–83, 2005.

KP06. H. Kalte and M. Porrmann. REPLICA2Pro: task relocation by bitstream manipulation in Virtex-II/Pro FPGAs. *Proceedings of the 3rd conference on Computing frontiers*, pages 403–412, 2006.

KZJS00. D. Keymeulen, R.S. Zebulum, Y. Jin, and A. Stoica. Fault-Tolerant Evolvable Hardware Using Field-Programmable Transistor Arrays. *IEEE Transactions on Reliability*, 49:305–316, 2000.

Lap85. J. Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. In *Digest of Papers FTCS-15: 15th International Symposium on Fault-Tolerant Computing*, pages 2–11 IEEE Computer Society Press, Los Alamitos, CA, Los Alamitos, CA, 1985.

LMSP99. J. Lach, W.H. Mangione-Smith, and M. Potkonjak. Algorithms for efficient runtime fault recovery on diverse FPGA architectures. In *DFT '99. International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 386–394, 1999.

Mas07. Jeff Mason. Research Engineer, Xilinx Labs, Logmont, CO. Personal Coorespondance. 3 January 2007.

Max04. C. Maxfield. *The Design Warriors Guide to FPGAs.* Academic Press, Inc., Orlando, FL, 2004.

McF94. C. McFarland. Computer Subsystem. 1994. http://www.tsgc.utexas.edu/archive/subsystems/.

MHS^{+}04. Subhasish Mitra, W.-J. Huang, N.R. Saxena, S.-Y. Yu, and E.J. McCluskey. Reconfigurable architecture for autonomous self-repair. *IEEE Design & Test of Computers*, 21(3):228–240, 2004.

MMP+03. D. Mesquita, F. Moraes, J. Palma, L. Möller, and N. Calazans. Remote and Partial Reconfiguration of FPGAs: tools and trends. *Proceedings of the 17th Parallel and Distributed Processing Symposium (IPDPS03)*, pages 1–8, 2003.

NAS00. NASA. Radiation Effects & Analysis: Single Event Effects. 2000. http://radhome.gsfc.nasa.gov/radhome/see.htm.

Nel90. Victor P. Nelson. Fault-Tolerant Computing: Fundamental Concepts. *IEEE Computer*, pages 20–25, 1990.

RS02. A.K. Raghavan and P. Sutton. JPG A Partial Bitstream Generation Tool to Support Partial Reconfiguration in Virtex FPGAs. *Proceedings of International Parallel and Distributed Processing Symposium. IPDPS 2002, Abstracts and CD-ROM*, pages 155–160, 2002.

SA04. N. Steiner and P. Athanas. An Alternate Wire Database for Xilinx FPGAs. In *Proceedings of the Twelfth Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2004*, pages 336–337, 2004.

SBB+06. P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular dynamic reconfiguration in Virtex FPGAs. *IEE Proceedings Computers and Digital Techniques*, 153(3):157–164, 2006.

Tor02. Jim Torresen. Reconfigurable Logic Applied for Designing Adaptive Hardware Systems. In *Proc. of the International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet (SSGRR2002W)*, 2002.

US05. Andres Upegui and Eduardo Sanchez. Evolving Hardware by Dynamically Reconfiguring Xilinx FPGAs, September 2005.

VN56. J. Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies, Annals of Math. Studies*, (34):43–98, 1956.

WB04. J. Williams and N. Bergmann. Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA04)*, 2004. Las Vegas, Nevada.

WSW06. J. Wu, I. Syed, and J. Williams. Creating a Simple uClinux ready MicroBlaze Design. 2006. www.itee.uq.edu.au/wu/downloads/uClinux_ready_Microblaze_design.pdf.

Xil04a. Xilinx. Dynamic Reconfiguration of RocketIO MGT Attributes. *XAPP660 (v2.2)*, 2004. http://www.xilinx.com .

Xil04b. Xilinx. OPB HWICAP. *DS 280 (v1.3)*, 2004. http://www.xilinx.com .

Xil04c. Xilinx. Two Flows for Partial Reconfiguration: Module Based or Difference Based. *XAPP290*, 2004. http://www.xilinx.com .

Xil04d. Xilinx. Virtex Series Configuration Architecture User Guide. *XAPP151*, 2004. http://www.xilinx.com .

Xil05a. Xilinx. Constraints Guide 8.1i. 2005. http://www.xilinx.com .

Xil05b. Xilinx. Virtex-II Pro and Virtex-II Pro X FPGA User Guide. *UG012*, 2005. http://www.xilinx.com .

Xil06. Xilinx. Early Access Partial Reconfiguration User Guide For ISE 8.1.01i. *UG208*, (UG208), 2006. http://www.xilinx.com .

Xil07a. Xilinx. Partial Reconfiguration Early Access software tools. 2007. http://www.xilinx.com/support/prealounge/protected/index.htm.

Xil07b. Xilinx. Virtex-5 and Virtex-4 Features. 2007. http://www.xilinx.com .

XSHL99. Jian Xu, Paifa Si, Weikang Huang, and F. Lombardi. A novel fault tolerant approach for SRAM-based FPGAs. In *Dependable Computing, 1999. Proceedings. 1999 Pacific Rim International Symposium on*, pages 40–44, 1999.

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

1. REPORT DATE *(DD–MM–YYYY)* 21–02–2007	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Aug 2005 — Mar 2007
4. TITLE AND SUBTITLE Using Relocatable Bitstreams For Fault Tolerance		**5a. CONTRACT NUMBER**
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Montminy, David P., Captain, USAF		**5d. PROJECT NUMBER**
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, Graduate School of Engineering and Management, 2950 Hobson Way, WPAFB OH 45433-7765		**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GCE/ENG/07-09
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/VSSE, Attn: Mr. Ken K. Hunt, Air Force Research Laboratory, 3550 Aberdeen Ave SE, Bldg 891, Kirtland AFB, NM 87117-5776 DSN 246-4959		**10. SPONSOR/MONITOR'S ACRONYM(S)**
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This research develops a method for relocating reconfigurable modules on the Virtex-II (Pro) family of Field Programmable Gate Arrays (FPGAs). A bitstream translation program is developed which correctly changes the location of a partial bitstream that implements a module on the FPGA. To take advantage of relocatable modules, three fault-tolerance circuit designs are developed and tested. This circuit can operate through a fault by efficiently removing the faulty module and replacing it with a relocated module without faults. The FPGA can recover from faults at a known location, without the need for external intervention using an embedded fault recovery system. The recovery system uses an internal PowerPC to relocate the modules and reprogram the FPGA. Due to the limited architecture of the target FPGA and Xilinx tool errors, an FPGA with automatic fault recovery could not be demonstrated. However, the various components needed to do this type of recovery have been implemented and demonstrated individually.

15. SUBJECT TERMS

FPGA, partial reconfiguration, fault tolerance

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Dr. Rusty Baldwin
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
U	U	U	UU	131	(937) 255–3636 x4445, rusty.baldwin@afit.edu