SECURE PARTIAL RECONFIGURATION
OF FPGAS

by

Amir H. Sheikh Zeineddini
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of the
the Requirements for the Degree
of
Master of Science
Electrical and Computer Engineering

Committee:

_____	Dr. Kris Gaj, Thesis Director
_____	Dr. Peter W. Pachowicz
_____	Dr. William Sutton
_____	Andre Manitius, Chairman, Department of Electrical and Computer Engineering
_____	Lloyd J. Griffiths, Dean, School of Information Technology and Engineering
Date: _____	Summer 2005 George Mason University Fairfax, VA

Secure Partial Reconfiguration of FPGAs

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Amir H. Sheikh Zeineddini
Bachelor of Science
Azad University, 2000

Director: Dr. Kris Gaj, Associate Professor
Department of Electrical and Computer Engineering

Summer 2005
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to my family for their endless love and support over the years.

To the person who always believed in me, my greatest supporter, best friend, and partner Haanieh Riahi.

Acknowledgments

First, I wish to thank my advisor, Dr. Kris Gaj for all of his guidance, encouragement, and patience. Without his never ending support this project would never have been completed. Also, I would like to thank my committee members, Dr. Peter W. Pachowicz, and Dr. William Sutton, for the time they provided and their helpful comments and suggestions.

Additionally, I would like to thank my fellow graduate students Deapesh Misra, Milind M. Parelkar, and Pawel Chodowiec, who provided invaluable support and suggestions throughout this process.

Finally, I would like to thank my friends Haanieh Riahi and Reza Shakoori for their help and support.

Table of Contents

List of Tables

List of Figures

Abstract

SECURE PARTIAL RECONFIGURATION OF FPGAS
Amir H. Sheikh Zeineddini
George Mason University, 2005
Thesis Director: Dr. Kris Gaj

SRAM FPGAs are configured by loading application-specific configuration data−the bitstream−into an internal configuration memory. Because the configuration memory is a SRAM volatile memory, it must be configured each time the device is powered up. The necessity of configuration on each power up makes it easier for attackers to clone, reverse engineer, or tamper the bitstream during configuration. Bitstream encryption is the most effective and practical solution to improve the security of SRAM FPGAs and protecting the configuration data.

The existing Xilinx solution uses CAD tools support for bitstream encryption and an on-chip special circuit for decryption. One of the drawbacks of this solution is that it is not possible to use partial reconfiguration and readback when the device is configured with an encrypted bitstream. Partial reconfiguration changes the design behavior in a portion of the FPGA without full reconfiguration by using a partial bitstream. Different forms of this new capability provide many advantages such as run-time reconfiguration for various application areas.

This thesis investigates a method to perform a secure partial reconfiguration and improve the security of SRAM FPGAs through exploiting a configuration controller

that enables an FPGA to dynamically reconfigure itself under the control of an embedded processor core. The hardware architecture of this configuration controller was implemented with a minimal footprint using two schemes: one based on hard-wired PowerPC processor core and the second based on MicroBlaze soft processor core. The software part of the controller consists of the program responsible for loading the configuration bitstream from external memory, authentication, decryption, and partial reconfiguration of the FPGA. This scheme enables embedded systems that benefit from partial reconfiguration to increase their design security without requiring external circuitry and provides flexibility by allowing the use of various authentication and encryption/decryption algorithms.

The scheme is tested for partially reconfigurable designs containing the configuration controller and an application system within a single FPGA. Comparison of the total configuration time (including authentication and decryption) and resource utilization targeting Xilinx Virtex-II Pro devices are also provided.

Keywords– Design Security, FPGA Bitstream, Dynamic Partial Reconfiguration, Platform FPGA

Glossary

AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
BRAM	BlockRAM
BUFGMUX	Global Multiplexed Buffer
CLB	Configurable Logic Block
DLMB	Data-side Local Memory Bus
DMA	Direct Memory Access
EDK	Embedded Development Toolkit
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input Output
Hard IP	A fixed IP on the FPGA fabric.
HDL	Hardware Description Language
HMAC	Hash Message Authentication Code
HWICAP	Hardware Internal Configuration Access Port
ICAP	Internal Configuration Access Port
ILMB	Instruction-side Local Memory Bus
JTAG	Joint Test Action Group
LMB	Local Memory Bus
LUT	Look-up Table
MHS	Microprocessor Hardware Specification
MicroBlaze	A 32-bit soft processor developed by Xilinx.
MIPS	Million Instruction Per Second
MMU	Memory Management Unit
MSS	Microprocessor software Specification
OPB	On-chip Peripheral Bus
PLB	Processor Local Bus
PowerPC	A type of RISC microprocessor developed jointly by Motorola, Apple and IBM.
RISC	Reduced Instruction Set Computer
Soft IP	A synthesizable Intellectual Property which can be readily incorporated into an FPGA.
TBUF	Tri-state Buffer
Throughput	The amount of information processed in a given time.
TLB	Translation Look-aside Buffer
UART	Universal Asynchronous Receiver Transmitter
Volatile	Memory in which data is lost when power is removed.
Watchdog Timer	A hardware timer that is periodically reset by software.
Word	16 bits
XMD	Xilinx Microprocessor Debugger
XPS	Xilinx Platform Studio

Chapter 1: Introduction

1.1 Overview

As the performance gap between FPGAs and ASICs decreases [1], platform FPGAs with various configurable elements and embedded blocks provide new solutions for high density and high-performance embedded system designs. These platforms not only enable system architects to design and develop complex custom systems using embedded processor and interoperable IP cores but also provide technologies such as dynamic reconfiguration of part of an FPGA while other areas of the device remain operational. There are many advantages in partial dynamic reconfiguration, especially for applications that require adaptive and flexible hardware such as mobile communication applications and real-time embedded systems. Deploying dynamic run-time reconfiguration in systems results in reduced chip area and power consumption.

Considering the wide range of features, platform FPGAs address many new application areas and therefore an increase in their popularity makes the need for design security mechanisms even more important especially in high-security areas where they might not otherwise be acceptable. The design security must protect the design against cloning and reverse engineering that correspond to different attacks. A survey in [2] analyzes possible attacks against FPGAs. In the case of SRAM FPGAs this is directly concerned with protection of bitstream especially during configuration and reconfiguration. Bitstream encryption as a solution increases the level of security and

makes the configuration bitstream secure against attackers.

The Xilinx [3] security solution (SecureChip) uses CAD tools for bitstream encryption and an internal circuit for decryption [4]. The major disadvantage of this scheme is that the partial reconfiguration capability of the FPGA is disabled and therefore a device configured with an encrypted bitstream cannot be partially reconfigured externally. By using new features of platform FPGAs we propose a method using a configuration controller to achieve bitstream security specifically for designs that benefit from partial reconfiguration. The controller is implemented in the form of a self-reconfiguring platform. Self-reconfiguration extends the dynamic reconfiguration capability by using particular circuits on the logic array such as embedded processor cores to control the configuration of other areas of the FPGA.

The configuration controller is capable of performing secure partial reconfiguration of the FPGA after the initial configuration and provides the flexibility of using arbitrary algorithms for authentication and encryption/decryption of partial bitstreams. This can also facilitate secure remote partial reconfiguration for vendor updates and feature upgrades in the field.

1.2 Method

In this thesis, the configuration controller is implemented using Xilinx Virtex-II Pro devices. Figure 1.1 shows the major components of the proposed scheme. The embedded processor in the configuration controller is able to partially reconfigure portions of an application system with encrypted IP cores. Embedded microprocessor reads an encrypted partial bitstream from an external memory to authenticate and decrypt it using software cores. To perform partial reconfiguration it uses a special configuration

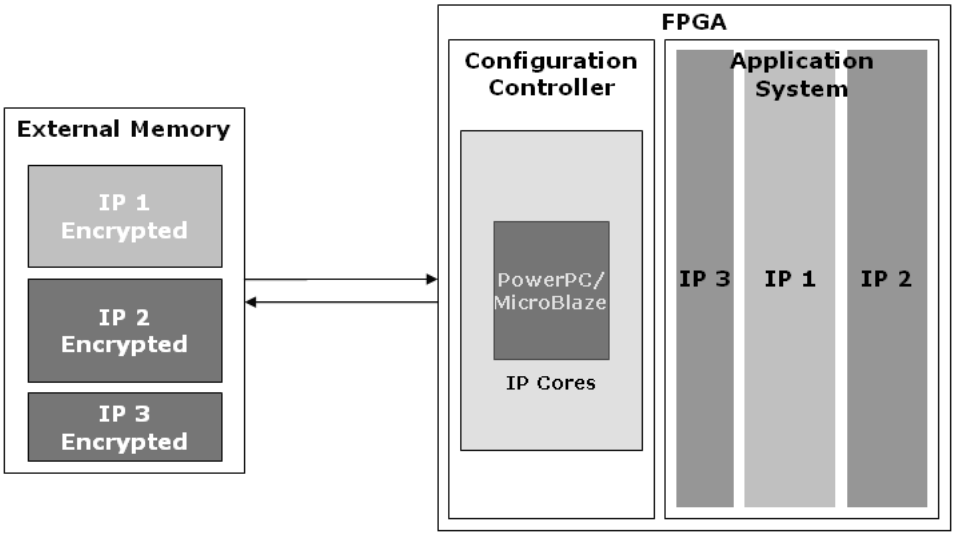


Figure 1.1: General Structure of the Scheme

interface called Internal Configuration Access Port (ICAP).

PowerPC hard and MicroBlaze soft embedded processors were separately used in constructing the hardware of the configuration controller. It should be noted that throughout this thesis we use the term configuration controller and the term self-reconfiguring platform interchangeably. To analyze and evaluate each of these self-reconfiguring platforms, they were assembled with an application system in partially reconfigurable system-on-a-chip designs using difference-based and module-based flows for partial reconfiguration.

Xilinx EDK and ISE provided the framework for design of hardware and software components. All designs targeted a Xilinx Virtex-II Pro FPGA on a Xilinx ML310 Embedded Development Platform.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 presents the related work and motivation. In chapter 3 an overview of Virtex-II Pro platform FPGA, partial reconfiguration, and EDK tools and ML310 evaluation board is presented. Chapter 4 explains the hardware architecture of the implemented self-reconfiguring systems for both hard/soft processor cores as well as the software architecture. Chapter 5 presents the methodology of the experiment. Chapter 6 presents the obtained results and the discussion of the results. Finally, chapter 7 provides conclusion for this thesis.

Chapter 2: Related Work and Motivation

2.1 FPGAs and Security

FPGAs are increasingly being used for many systems and efficient SoC (System-on-a-Chip) designs. Competitive market environment and high security areas such as military systems are among the factors that make protecting designs implemented in FPGAs more important. Without proper safeguards, design information and proprietary intellectual properties face major security risks and attackers will be able to steal the design contained in the bitstream of FPGAs. Common approaches to design theft are:

- *Cloning* – In cloning an attacker makes an exact copy of the design bitstream or layout without necessarily understanding the details of implementation. It is considered as the primary form of IP theft.

- *Reverse engineering* – This form of piracy involves analyzing the configuration file and then reconstructing it in HDL/RTL/netlist representation.

Reverse engineering is more complex than cloning because it requires an attacker to have technical expertise to understand how the design works. It is also considered to be a more serious threat since the design can be modified or improved afterwards.

Another form of security risk involves tampering the design with malicious intent and replacing it with a harmful design capable of damaging the device or stealing the

sensitive information. It is a particular concern in high-security areas such as military and financial applications.

Different types of FPGAs provide various levels of protection against these attacks. Currently FPGAs are made based on three different technologies:

- *SRAM-based FPGAs* – Based on a volatile memory technology, these devices must be initialized or configured on each power-up typically by loading a bit-stream from an on-board/external configuration device.

- *Antifuse FPGAs* – These devices are one-time programmable and their functions are immediately available upon system power-up. They are programmed by creating a link between the two terminals of each antifuse node using a high voltage.

- *Flash FPGAs* – The connections in Flash FPGAs are realized through flash transistors [2]. These devices are programmed by changing the charge present on the floating gates of the flash transistors.

The non-volatile nature of flash and antifuse FPGAs makes them more secure and practically resistance against different attacks because it is extremely difficult, cost-inefficient, and time-consuming to copy or reverse engineer these devices. On the other hand, SRAM-based FPGAs are volatile and the necessity of configuration on each power up makes it easier for attackers to clone, reverse engineer, or tamper the bitstream during configuration. Table 2.1 summarizes resistant and vulnerability of different types of FPGAs against the attacks.

However, reconfigurability, flexibility, and high-density of SRAM-based FPGAs results in a higher market share comparing to other types of FPGAs and therefore

Table 2.1: THE RESISTANCE OF FPGAS AGAINST VARIOUS ATTACKS

Types of Attacks	Types of FPGAs		
	SRAM	Antifuse	Flash
Cloning	Vulnerable	Resistant	Resistant
Reverse Engineering	Vulnerable	Resistant	Resistant
Tampering	Vulnerable	Resistant	Vulnerable

improving the security level of such FPGAs is the focus of research among the manufacturers and research community.

2.2 Bitstream Encryption

Bitstream encryption is the most effective and practical solution to improve the security of SRAM-based FPGAs and protecting the configuration file. Bitstream encryption prevents any access to the content of the bitstream without the knowledge of the secret keys. The keys are stored in a volatile or non-volatile location inside the FPGA. During configuration, these stored keys are used for decrypting the encrypted bitstream stored in an external memory.

It should be noted that even though bitstream encryption makes SRAM-based FPGAs more secure against cloning and reverse engineering and currently is the only implemented solution, security should also be augmented with authentication as a way to prevent tampering the bitstream by an active attacker with malicious intent.

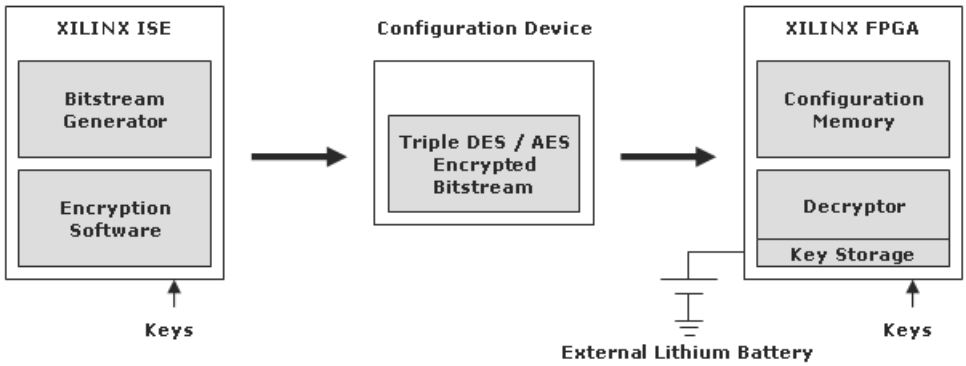


Figure 2.1: Xilinx SecureChip Technology

In the following sections some implemented and proposed schemes for bitstream encryption are provided.

2.2.1 Xilinx SecureChip Technology

The Xilinx SecureChip technology is simple and efficient. All Virtex II family devices (Virtex-II, Virtex-II Pro, and Virtex-II Pro X FPGAs) use the Triple DES encryption scheme [4]. In Virtex-4 devices Triple DES has been replaced with AES to increase security and throughput. The scheme exploits software support of Xilinx ISE CAD tools for both encryption of the bitstream and key generation. Figure 2.1 shows the Xilinx security system.

For decryption, it uses an on-chip decryptor along with the internal decryption keys stored in a dedicated memory. Either externally-connected battery or an auxiliary power supply (VCCAUX) is the source of power for volatile storage of the keys. The keys are erased if there is a tampering with the device.

The problem with this scheme is the extra area and cost needed for the external

battery, lack of flexibility, and the disablement of partial reconfiguration and readback for encrypted bitstreams.

2.2.2 Other Proposed Solutions

A method proposed by Algotronix [6] removes the need for an external battery by finding another way of storing the secret key on the FPGA such as use of laser to engrave the key. This will make it necessary for the FPGA to contain both encryption and decryption circuits and hence there is no need for the software to support encryption. This solution uses even more FPGA silicon area than Xilinx scheme and it also lacks flexibility since the encryption and decryption circuits are fixed with no possibility of upgrade or use of another algorithm. Also since the key is stored on the FPGA each chip is set up with a different random key and it cannot be changed after manufacture.

In [7] a new solution is proposed with no implementation available at the moment. The scheme selects and places the cores for encryption and decryption in the FPGA and then removes them to free the chip area. A dedicated configuration controller manages both the encryption and decryption configuration schemes by relying on partial and self reconfiguration. This configuration scheme also uses an embedded key instead of an externally-powered storage for the secret key.

Figure 2.2 shows an example of the decryption management in this system. In this example the application is partitioned into three different parts. Two parts require high security and are encrypted using two different encryption algorithms and one part requires no encryption.

Before decryption step, first, the FPGA is configured with the encryption circuit

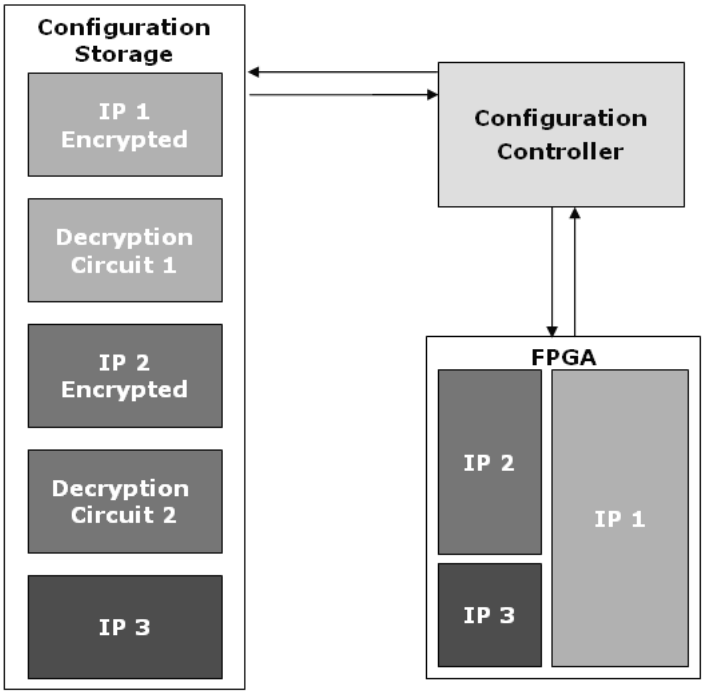


Figure 2.2: Dedicated Configuration Controller Scheme

of IP1 and IP2 in order to encrypt the configuration of these parts. Then all the encrypted configurations along with the non-encrypted configuration for IP3 and the configuration of the decryption circuits required to decrypt IP1 and IP2 are stored in a configuration storage.

The configuration process in the decryption phase works as follows: First the FPGA is configured with the decryption circuit 1 for decrypting the encrypted configuration of IP1. Then the FPGA auto-configures the IP1 inside the FPGA and replaces the decryption circuit 1 with the decryption circuit 2 in order to decrypt the encrypted configuration of IP2 and auto-configure the IP2. The last step consists in configuring the FPGA free area with non-encrypted configuration of IP3.

The method is flexible and adjusts the security level to application needs but is

relatively complex considering the limitations for partial reconfiguration imposed by the FPGA manufacturers and the CAD tools.

2.3 Self-reconfiguring Platform

The idea of implementing a self-reconfiguring platform for Xilinx Virtex family was first reported in [5]. The platform enabled an FPGA to dynamically reconfigure itself under the control of an embedded microprocessor. The provided hardware architecture established the framework for the implementation of the self-reconfiguring platforms in this thesis. The authors also presented the first detail description of the two core software components developed by Xilinx which remove the low level details of the configuration interface. Internal Configuration Access Port Application Program Interface (ICAP API) and Xilinx Partial Reconfiguration Toolkit (XPART) provided methods for reading and modifying select FPGA resources and support for relocatable partial bitstreams.

The presented methods for hardware architecture are lightweight (minimal area) and improvable as planned by the authors. The more efficient implementation moves parts of the control logic for ICAP API implemented in software to hardware core. The XPART software layer is only reported and not currently available in the Xilinx CAD tools and hence the application developed in this thesis only employed the ICAP API.

Chapter 3: Background

3.1 Virtex-II Pro Platform FPGA

3.1.1 Architecture

Overview

The Virtex-II Pro Platform FPGA is the first product of a new paradigm shift from programmable logic to programmable systems. It is capable of implementing flexible, high performance, and low cost system-on-a-chip designs by combining a variety of embedded features with specially developed hardware/software IP cores [8].

The architecture incorporates both embedded RocketIO Multi-Gigabit Transceivers (MGTs) that create high-speed serial links between devices and a hard processor core within the FPGA fabric. High I/O bandwidth and high performance general purpose processor cores are especially beneficial for networking applications, DSP systems, and deeply embedded systems.

Figure 3.1 shows the generic architecture of Virtex-II Pro FPGA family. Virtex-II Pro is processed at 0.13 μ and except for the unique features of its architecture all other FPGA features are identical to Virtex-II devices.

The next sections describe some of the important functional components of the Virtex-II Pro Platform FPGA architecture related to this study.

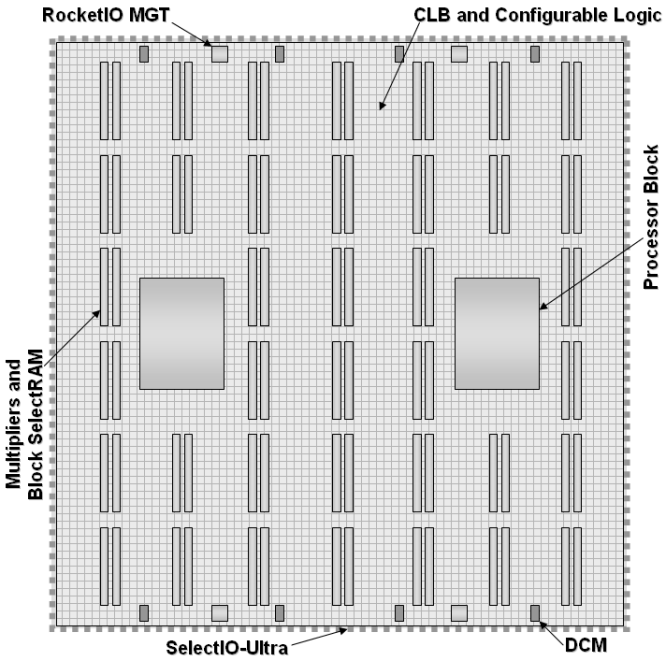


Figure 3.1: Virtex-II Pro Generic Architecture Overview

Processor Block

Figure 3.2 shows the internal architecture of the Processor Block containing four components: Embedded IBM PowerPC 405-D5 RISC CPU core, On-Chip Memory (OCM) controllers and interfaces, Clock/control interface logic, and CPU-FPGA Interfaces.

The OCM controllers are special instruction and data memory interfaces that support the attachment of additional memory, the BRAMs in FPGA fabric, to the instruction and data caches within the PowerPC core. OCM memory can be accessed at performance levels matching the cache arrays and provides access to optional, user-configurable direct-mapped memory. Typical applications of OCM memory include storage of interrupt service routines and scratch-pad memory.

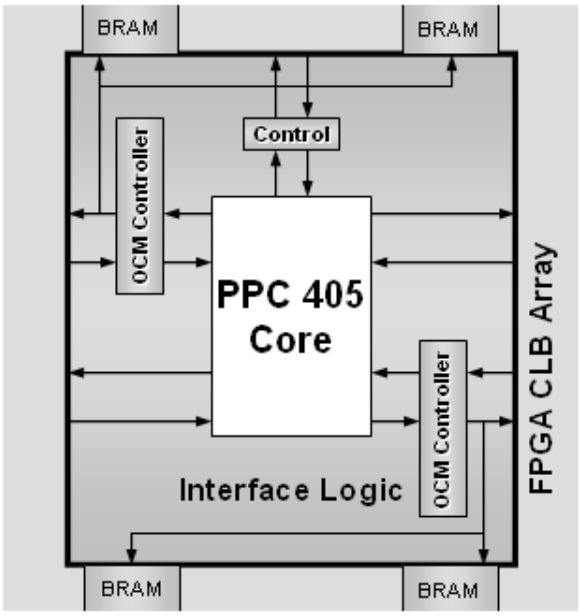


Figure 3.2: Processor Block Architecture

The Clock/control interface logic provide connectivity for the processor clock similar to CLB clock pins. Therefore the processor clock source can come from DCM, CLB, or user package pin.

All Processor Block user pins connect to the general FPGA routing resources through the CPU-FPGA interface. The processor block has the same routing resources available as other user signals. The CPU-FPGA interface provides connectivity for the following interfaces:

- *PPC405 core Processor Local Bus (PLB)*
- *PPC405 core Device Control Register (DCR) Bus*
- *On-Chip Memory (OCM)*
- *Reset and Debug*

- *Clock/Power Management (CPM)*
- *External Interrupt Controller (EIC)* – Presents external interrupts to the PPC405 core.

Embedded PowerPC 405 Core

The Processor Block incorporates fully embedded IBM PowerPC 405 processor core which is an implementation of the PowerPC embedded environment architecture [9]. The PowerPC architecture is a 64-bit architecture with a 32-bit subset. The embedded PPC405 core is a 32-bit Harvard architecture processor that operates in a five-stage pipeline and most instructions execute in a single cycle. The architecture is big endian internally and contains a virtual-memory-management unit that supports multiple page sizes. It is capable of more than 300 MHz clock frequency and 420 Dhrystone MIPS. The embedded PPC405 core provides the following functional blocks:

- *Cache units* – Allow concurrent accesses and minimize pipeline stalls. The instruction and data cache array are 16 KB each. Both cache units are two-way set associative. The PPC405 core accesses external memory through the 64-bit PLB master interface of cache units.
- *Fetch and Decode Logic* – Maintains a steady flow of instructions to the execution unit by placing up to two instructions in the fetch queue. It also examines the branch instructions to facilitate static branch prediction.
- *Execution Unit* – Contains the register file comprised of thirty-two 32-bit general purpose registers (GPR), ALU, and the multiply-accumulate (MAC) unit. The execution unit performs all 32-bit PowerPC integer instructions in hardware.

- *Memory Management Unit (MMU)* – Provides address translation of a 4 GB address space by using a TLB with 64 entries, protection functions, and storage attribute control for embedded applications. The MMU supports demand-paged virtual memory.

- *Timers* – The embedded PPC405 core contains a 64-bit time base and three timers: Programmable Interval Timer (PIT), Fixed Interval Timer (FIT), and Watchdog Timer (WDT) [9]. The PPC405 also provides an interface to an interrupt controller that is logically outside the PPC405 core.

- *Debug Logic* – Provides access to the resources on the embedded PPC405 core using supported tools depending on the debug mode: ROM monitors, JTAG debuggers, and instruction trace tools.

3.1.2 Configuration

Process and Flow

Configuration is the process of loading the configuration bitstream, a series of configuration commands and application-specific data, into the FPGA internal configuration memory. Configuration memory is arranged in a rectangular array of bits. One-bit wide vertical frames are the smallest addressable segments of the Virtex-II Pro configuration memory space [10]. Frames stretch from the top edge of the device to the bottom and have different sizes based on the device. Data is loaded on a column-basis and each column contains multiple frames. Different column types correspond roughly to physical device resources. All Virtex-II Pro devices have the same configuration column types: IOB, IOI, CLB, GCLK, BlockRAM, and BlockRAM Interconnect.

The Virtex-II Pro configuration control logic consists of a packet processor, a set of registers, and global signals that are controlled by the configuration registers. The packet processor controls the flow of data from the configuration interface to the appropriate register. The registers control all other aspects of configuration.

Bitstream is delivered through one of the configuration interfaces (JTAG, SelectMAP, or Slave/Master Serial) in a specific sequence based on the selected configuration mode. Writing or reading some or all of a configuration is done by issuing configuration commands to the desired interface followed by the configuration data.

There are four major phases in the configuration process: clearing configuration memory, initialization, bitstream loading, and device startup. After clearing the configuration memory the mode pins are sampled in the initialization phase and the configuration process begins. The target FPGA starts to receive data frames in the bitstream loading phase. This process is similar for all configuration modes; the primary difference between modes is the interface to the configuration logic. Device startup phase is a transition phase from the configuration mode to normal programmed device operation. The Start-Up Sequencer is an 8-phase sequential state machine that counts from phase 0 to phase 7. Upon completion of the start-up sequence, the target FPGA is operational.

Configuration Interfaces

Configuration interface is a logical interface consisting of one or more special configuration pins. Each configuration interface corresponds to one or more configuration modes. These modes are selectable via mode pins:

- *Master/slave Serial Mode* – In serial configuration mode, the FPGA is configured by loading one bit per CCLK cycle. The Slave Serial configuration mode allows for FPGAs to be configured from other logic devices such as microprocessors, or in a daisy-chain fashion. FPGAs CCLK pin is driven by an external source. In Master Serial mode, the FPGA drives the CCLK pin and it can be configured from a Serial PROM.

- *Boundary-Scan Mode* – The Boundary Scan (JTAG) interface allows bit-serial access to the configuration. It is a permanent interface that is always present.

- *SelectMAP Mode* – This mode is the fastest configuration option and provides an 8-bit bidirectional data bus interface to the configuration logic. The SelectMAP interface is typically driven by a processor, microcontroller, or some other logic device such as an FPGA or a CPLD. Multiple devices can also be chained in parallel.

Internal Configuration Access Port (ICAP)

The Virtex-II Pro configuration architecture features an Internal Configuration Access Port (ICAP) that provides the user logic with access to FPGA configuration interface and therefore direct access to memory bits of configuration memory [10]. The interface is similar to SelectMAP interface but without the restrictions on the readback and reconfiguration in case of encrypted bitstream. It does not support different configuration modes and cannot be used for full configuration. With no handshaking mechanism ICAP interface can be clocked up to the maximum frequency of 66MHz [5]. The ICAP block exists in the lower-right corner of the logic array.

3.2 Dynamic Partial Reconfiguration

3.2.1 Introduction

FPGA devices are partially reconfigured by loading only a subset of configuration frames into the FPGA internal configuration memory. The Xilinx Virtex-II Pro FPGAs allow partial reconfiguration in two forms: static and dynamic.

Static (or shutdown) partial reconfiguration takes place when the rest of the device is inactive and in shutdown mode. The non-reconfigurable area of the FPGA is held in reset and the FPGA enters the start-up sequence after partial reconfiguration is completed. In contrast, in dynamic (or active) partial reconfiguration new data can be loaded to dynamically reconfigure a particular area of FPGA while the rest of it is still operational. User design is not suspended and no reset and start-up sequence is necessary.

The general reason for partial reconfiguration is changing the design behavior without full reconfiguration. Dynamic partial reconfiguration has additional advantages when runtime-reconfiguration and efficient resource utilization is desirable. Runtime-reconfiguration is especially useful for applications that require adaptive and flexible hardware since they need to change the behavior of a system to adapt it to externally changing surroundings. Dynamic partial reconfiguration also facilitates more efficient changing configurations which results in reduced chip area and power consumption and availability of more FPGA resources.

Combined with other features, a more advanced form of reconfigurability can be realized when specific circuits on the FPGA control the partial reconfiguration. Chapter 4 describes this form of self-reconfiguration in detail.

A partially reconfigurable design consists of a set of full designs and partial modules. The full and partial bitstreams are generated for different configurations of a design. Partial bitstreams are subsets of a complete bitstream. Depending on the granularity of reconfiguration, partial bitstreams are used to make small content changes of particular FPGA logic elements such as BRAMs or to configure a large portion of the FPGA logic area implementing a new module or replacing an existing one. The partial bitstreams configure full columns of the FPGA. The sections of the column that were not modified are reconfigured with the same data. Because the FPGA memory cells have glitchless transitions, if the reconfiguration is performed on a frame basis, as supported by the Virtex family architecture, unmodified logic will continue to operate unaffected. Two exception for this rule are LUTs configured in Shift Register Mode or as a RAM. They will lose their state during readback or modification.

Currently there is no simple methodology to implement partially reconfigurable designs and there is limited support and automation of implementation tools to generate appropriate partial bitstreams. A number of academic and commercial groups are working on projects aiming at developing more efficient tools and methodologies [11].

The next sections will focus on software flows and Xilinx recommended methodologies for partial reconfiguration using current Xilinx CAD tools and devices [12]. These styles can be applied to different systems by the degree to which they utilize reconfiguration of the logic resources.

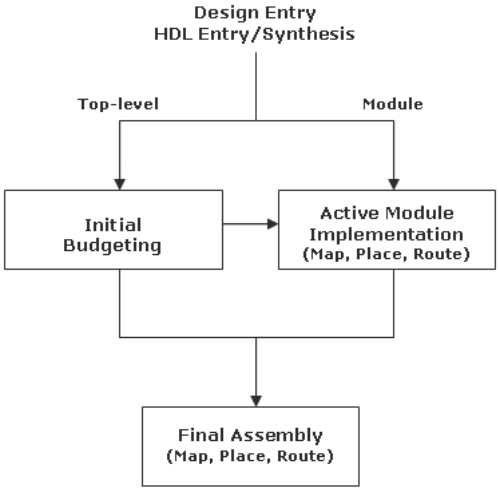


Figure 3.3: Modular Design Flow

3.2.2 Module-Based Partial Reconfiguration

Modular Design Overview

Module-based flow is suitable for partially reconfiguring a large portion of the design and is based on the Xilinx Modular Design methodology [13]. Modular Design allows large designs to be partitioned into self-contained modules that can be developed in parallel and independently to save time. Later, implemented modules are merged into one complete FPGA design. Figure 3.3 shows the overview of this flow. Similar to the standard design flow this flow comprises the following steps:

Design Entry and Synthesis Both the top-level design and modules are created using an HDL (Verilog or VHDL) or any other established design entry method. To synthesize them Xilinx synthesis tool, Xilinx Synthesis Technology (XST), can be used. This tool produces a netlist in NGC format.

Design Implementation This step consists of the three phases:

- *Initial Budgeting* – Creating the constraints and floorplan for the top-level design.
- *Active Module Implementation* – Implementing the top-level design with one module expanded at a time.
- *Final Assembly* – Assembling the top-level design and all implemented modules into a complete design.

Module-based partial reconfiguration flow is a modified version of modular design flow [12] that requires specific rules for reconfigurable modules and inter-module communications.

Reconfigurable Module Properties

Distinct portions of an FPGA, reconfigured by a partial bitstream, are referred to as reconfigurable modules. Certain properties and restrictions apply to their available resources, boundaries, and communication with other modules.

The resources in the area occupied by a reconfigurable module include slices, BRAMs, IOBs at any edge of the module, routing resources, TBUFs, and multipliers. All these resources are part of the module bitstream except for the clock resources which have separate configuration frames.

The boundary of a reconfigurable module is fixed and remains unchanged at all time. The height of the module must always be the full height of the device. The number of slice columns encompassed by the width of the module is a multiple of four

and the minimum width is four slice columns. Also, placement of a reconfigurable module is at a slice column which is a multiple of four [0, 4, 8, ...].

The states of storage elements (FFs and RAMs) inside the reconfigurable module are preserved during the reconfiguration process. It can be used as a 'prior state' information in the design. On the other hand, global set/reset (GSR) logic of FPGA cannot be used for initializing the state of a reconfigurable module.

Partial reconfiguration requires reconfigurable modules to use fixed communication channels (bus macros) to communicate with other modules both fixed and reconfigurable. Bus macros are used to establish unchanging communication paths between or through reconfigurable modules.

Bus Macro Communication

Module-based partial reconfiguration flow is further decomposed depending on whether communication is needed between the modules. A special bus macro is required for inter-module signals because signals connecting the reconfigurable modules with other modules can be routed differently in alternate design implementation. In other words the routing resources used for such signal must not change when the module is reconfigured. Bus macros are pre-routed hard macros that use fixed routing resources with no variation in routing in different design implementations.

Bus macros are currently implemented using 3-state buffers (TBUFs) in CLB tiles [12]. Figure 3.4 shows a bus macro used for inter-module communication and its implementation with TBUFs. At the dividing line between the boundary of two modules, one TBUF at each side is connected to other side via their output longlines horizontally. Based on the wiring of enable signals of bus macro the direction of each

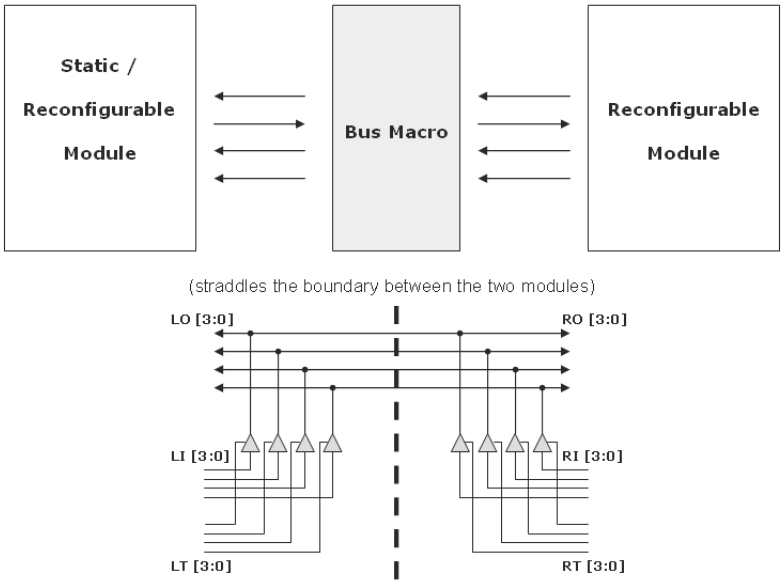


Figure 3.4: Physical Implementation of a 4-bit Bus Macro by Xilinx

one bit path can be from left-to-right or right-to-left. Only if bus macro is in leftmost position, bits 2 and 3 cannot go right-to-left and if bus macro is in rightmost position, bits 0 and 1 cannot go left-to-right. Position of a bus macro is locked and is placed in such way that it straddles the boundary line or area between two modules.

The number of horizontal longlines available in each slice row, four in Virtex-II Pro, is the limit to create fixed bridges between two modules. Xilinx will introduce a new implementation with the release of ISE 8.1 since Virtex-4 devices do not contain TBUFs.

Design Entry and Synthesis

HDL coding and synthesis process follow some general guideline in terms of the structure of top-level design, instantiation of bus macros, shared signals, and synthesis attributes.

All functional modules in the top-level design are instantiated with 'block-box' synthesis attribute. Only top-level should be synthesized with I/O buffer insertion enabled and there is no instantiation of I/Os inside the modules.

There is no additional logic in top-level except for modules and bus macros instantiation, I/O, and clocking logic. All defined clocks must use dedicated global routing resources and it is recommended to keep the clock design simple.

Modules should be self-contained with port definition clearly declared. There should be no shared signal other than the clock between the modules. This includes resets, constants (VCC, GND), enables, etc. Unlike a standard modular design, a partially reconfigurable design does not have intermodule ports and there are no pseudo-drivers or pseudo-ports.

As many as bus macros should be instantiated based on the number of bits needed for one reconfigurable module to communicate with other modules. Signals passing through reconfigurable modules connecting modules on the sides should be connected with bus macros and an intermediate signal in the reconfigurable module. During partial reconfiguration this signal cannot be actively used. However, a custom bus macro different from the one implemented by Xilinx that spans the width of the area between the modules can be used with no intermediate signal in the reconfigurable areas. This way the communication pass could be active during reconfiguration.

The static module(s) of the design should consider the "transition time" during reconfiguration when they rely on the state of the signals connected to reconfigurable module. Proper handshaking may be required.

Implementation Flow

The implementation flow takes place in three phases after the design entry.

Initial Budgeting In this phase, the design is floor planned and constrained based on the properties of each module. This includes the area-based floorplan of each module considering the boundary properties of reconfigurable modules. All top-level logic (IOBs and all global logic) should have fixed location constraints. Fixed location constraints for bus macros should be inserted using LOC constraints. Global-level timing constraints are also created in this phase. The result is a file with extension '.ucf' that is used for active implementation phase. There are various methods for entering constraint such as using Xilinx GUI tools (Constraint Editor, Floorplanner, ...) or adding constraints directly in VHDL/Verilog code.

Active Implementation This phase places and routes each module separately in the context of the top-level logic and constraints. Before running `ngdbuild`, `map`, and `par` for each module, any module-specific constraints is added to the constraints file created during the initial budgeting phase. After these steps BitGen is used to create partial bitstream for reconfigurable module and PimCreate publishes each module to be used in the next phase.

Final Assembly This phase combines all the placed and routed modules generated from the previous phase into a complete FPGA design. To maintain the performance of each module, placement and routing for each module are preserved.

At present, bitstreams generated for the full design require that the initial bitstream include at least one variation of any partially reconfigurable module. This

means that the initial bitstream should be a complete design since all global resources such as clocking logic need to be placed and properly constrained. Bitstream frames for global clocks are separate from other frames. This property enables partial reconfiguration to keep clock functional during reconfiguration but imposes a limit in which a completely separate module cannot be added to an initial design with partial reconfiguration flow.

3.2.3 Difference-Based Partial Reconfiguration

Using this flow the design can change either at the front-end or the back-end. For changes in HDL code or schematics at the front-end, the design must be re-synthesized and re-implemented while for back-end changes the FPGA Editor tool can modify sections of the design. Many different types of changes can be made using this tool including routing information, LUT programming, changing BRAM contents and I/O standards.

Back-end changes require understanding of how to make logic changes using the FPGA Editor tool, and the relevant options to select in BitGen. If the scale of changes in the design is small efficient use of this tool avoids re-synthesizing and re-implementing the design. The support of scripting in the FPGA Editor can also save time. The FPGA Editor GUI is able to open a routed design with '`.ncd`' extension and make the file available for modification. It is possible to select an individual slice and change the LUT equations or modify the contents of a BRAM displayed in the format of INIT synthesis constraint. Other changeable elements that can be modified in FPGA Editor include I/O standards, muxes that invert polarity, flip-flop initialization and reset values, or pull-ups/pull-downs on external pins.

The bitstream generator BitGen used with `-r` switch can create a partial bitstream that contains only the difference between the modified design and the initial bitstream. In other words, BitGen produces a partial bitstream that only configures the frames that are different between the two designs. The produced partial bitstream is small and quick to load since -r switch uses multiple write feature of BitGen and therefore generates a compressed bitstream. It should be noted that the bit length and reconfiguration speed of a partial bitstream are directly proportional. A partial bitstream can be loaded only after the device power up and loading an initial bitstream. For active partial reconfiguration the `-g ActiveReconfig:Yes` switch is also required to remove the shutdown commands from the partial bitstream. When using the SelectMAP interface for partial reconfiguration, the `-g Persist:Yes` switch is required as well.

3.2.4 Other Minor Software Flows

The BitGen "Partial Mask" flow allows users to select the configuration columns to be included in a partial bitstream. Different type of columns such as IOB, IOI, BRAM, or CLB columns can be selected with `-g` switch and proper PartialMask settings. A hex mask field for each PartialMask settings indicates exactly which columns to be written in a partial bitstream. This flow can only produces partial bitstreams for active partial reconfiguration and therefore it must be used with `-g ActiveReconfig:Yes` switch. The details of this flow is provided in [10].

Another feature is BlockRAM "Savedata" option that prevents writes to BRAMs during shutdown reconfiguration. This option can be set using FPGA Editor as described in [12]. This option is only safe to use with static partial reconfiguration

since it can interfere with BRAM operation during active reconfiguration.

3.3 Xilinx Embedded Development Kit (EDK)

3.3.1 Tool Architecture Overview

EDK provides a framework for design of hardware/software components of the embedded processor systems on programmable logic [14]. Appropriate tools for each stage of the design in addition to IBM PowerPC and Xilinx MicroBlaze processor cores infrastructure and peripheral IP cores facilitate hardware/software partitioning, design reuse, and lower time-to-market.

Embedded system tools in EDK consist of Xilinx Platform Studio (XPS), GNU software development tools, board support packages, and complete operating systems such as Wind River VxWorks, MontaVista Linux, and Xilinx MicroKernel.

3.3.2 Tool Flows

Figure 3.5 provides an overview of the tools flow. In a typical design of an embedded processor system, the first step is to create a hardware platform followed by the creation of software platform and optionally verification platform. Platform Studio, a GUI technology, with its underlying tools integrates all the processes from design entry to design debug and verification [15].

Creating a basic hardware system involves assembling a system containing processor, buses, and peripherals, generating an HDL netlist, and implementing the design using ISE implementation tools to generate a bitstream. The hardware platform is defined by the Microprocessor Hardware Specification (MHS) file. XPS GUI or Base

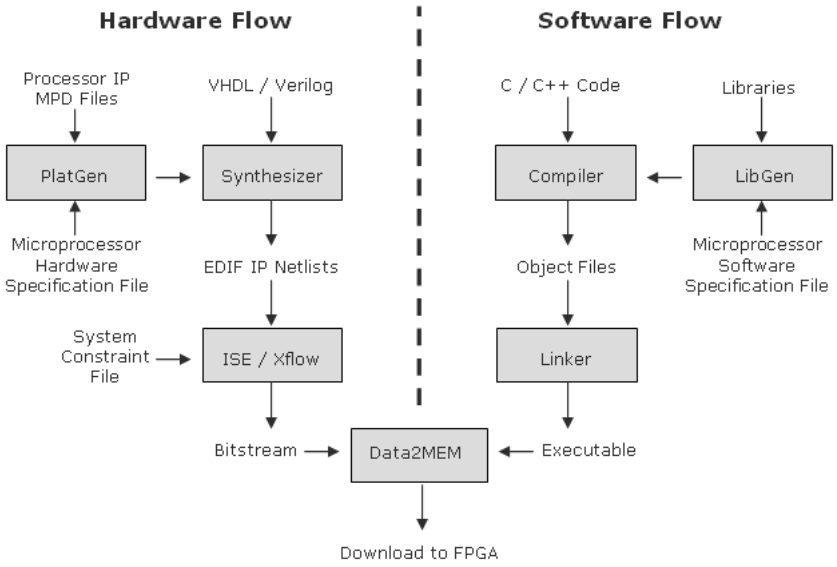


Figure 3.5: EDK Tools Flow

System Builder (BSB) can facilitate the creation of MHS file. The BSB provides a simple, highly automated method for creating an embedded design. To add or import a user peripheral Create and Import Peripheral Wizard can be used. Platform Generator (PlatGen) customizes and generates the HDL netlist using MHS file. Simulation Model Generator (SimGen) also uses MHS file to simulate and configure various VHDL and Verilog simulation models for the specified hardware.

Creating the software platform involves building libraries, compiling C applications, initializing bitstreams with the application, downloading applications onto external memories, and debugging applications using debugger. The software platform is defined by Microprocessor Software Specification (MSS) file. The MSS defines the OS, drivers for IPs, and other libraries. MSS file is generated by XPS using the specified software setting. Library Generator (LibGen) takes the MSS file as an input to configure libraries, device drivers, file systems, and interrupt handlers. XPS calls

GNU compiler tools provided for both hard/soft processors for compiling and linking user application executables. The Bitstream Initializer (BitInit) tool can then initialize the bitstream with the executable in the instruction memory of processors on the FPGA.

The bitstream can be downloaded using Xilinx Microprocessor Debugger (XMD), bootloader programs, or System ACE controller. XMD is the underlying engine to communicate to processor targets and provides an interface for both hardware system debug and software running on hardware. This tool is used with GNU debugger for software debugging.

3.4 Xilinx ML310 Evaluation Board

The ML310 Embedded Development Platform is a Virtex-II Pro based platform suitable for rapid prototyping and system verification. The main features of ML310 include: 256 MB DDR memory, System ACE CF controller, FPGA UART, GPIO LEDs/LCD, PCI bus interface, CPU debugging interfaces, SPI EEPROM, and high speed I/O through RocketIO Multi-Gigabit Transceivers. Figure 3.6 shows the block diagram of the ML310 board.

MGT blocks available in the Virtex-II Pro are flexible parallel-to-serial and serial-to-parallel embedded transceivers used for high-bandwidth interconnection between buses, backplanes, or other subsystems. The high-speed I/O signals on the FPGA are accessible through two personality module (PM) connectors on the ML310 board. The majority of the ML310 features are accessed over the 33 MHz/32-bit PCI bus. The Virtex-II Pro PPC405 processors can gain access to the primary PCI bus through the EDK PCI Host Bridge IP. The peripherals are either directly connected to the

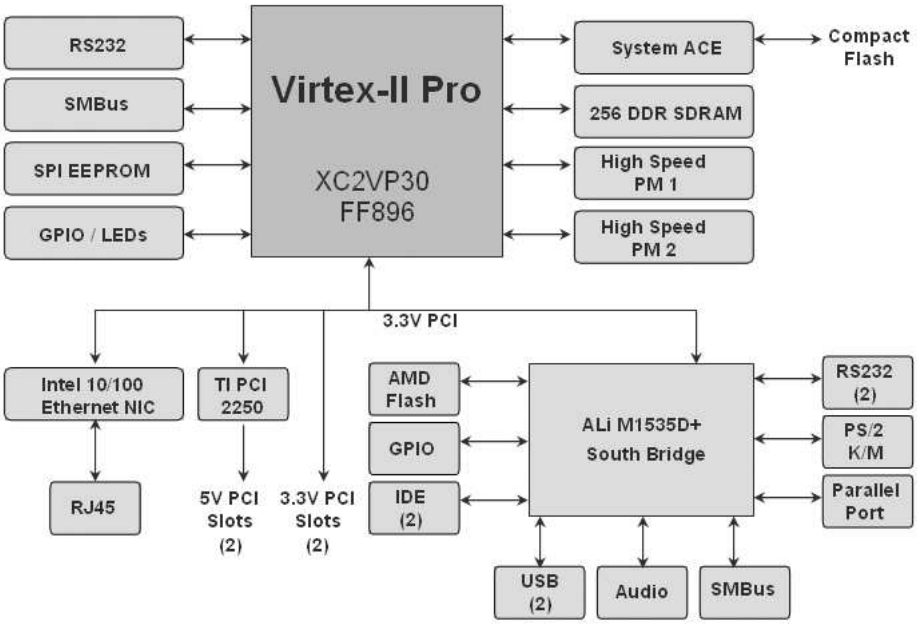


Figure 3.6: Block Diagram of Xilinx ML310 Embedded Development Board

FPGA or indirectly accessible by way of the PCI bus. The PCI bus is connected to fixed PCI devices such as Intel 10/100 PCI Ethernet NIC and ALi PCI South Bridge.

The ALi South Bridge augments the ML310 with many of the basic features found on legacy PCs such as parallel/serial/USB ports, IDE connectors, and GPIO. The main system clock of ML310 is a 100 MHz oscillator. The system clock is typically used to generate multiple clocks with varying frequency and phases within the FPGA fabric by using the Virtex-II Pro Digital Clock Managers (DCMs). The FPGA also generates and drives clocks required by the DDR memory and PCI bus interfaces.

Chapter 4: Implementation Methodology

4.1 Overview

Self-reconfiguration is an advanced form of configuration in which specific circuits on the FPGA are used to control the partial reconfiguration of a subset of the FPGA resources while the rest of device maintains correct operation. To create a self-reconfiguring platform the device must be dynamically reconfigurable. Also it is desirable that the device provides internal access to configuration port [5]. Xilinx Virtex-II Pro devices support both features.

An embedded processor can be used as a configuration controller in a self-reconfiguring platform. It provides the following advantages:

- Reduces the overall system complexity since fewer discrete devices are required for reconfiguration.
- Minimizes the latencies associated with accessing the configuration port since the control logic is as close to logic array as possible.
- Provides more reconfiguration options to the designer since the processor could manipulate (encryption, compression) the data before reconfiguring the device.

A Self-reconfiguring platform allows an application to get reconfiguration data from any peripheral such as a remote network or an external memory. It could also

help the OS to manage hardware tasks. The next sections describe the design of two self-reconfiguring platforms capable of performing dynamic partial reconfiguration of the FPGA under the control of embedded processor cores.

4.2 Design Description

In general, to design an embedded processor system, hardware components, memory map, and software application are needed. The hardware components of the constructed self-reconfiguring platforms target both embedded PowerPC hard processor core and MicroBlaze soft processor core. These platforms demonstrate how an embedded processor and the software running on it can be used as a configuration controller capable of performing a secure partial reconfiguration of the FPGA after the initial configuration. The application running on the embedded processor allows the processor to read the partial bitstream from an external memory, authenticate the signed partial bitstream, decrypt the encrypted partial bitstream, and dynamically reconfigure part of the FPGA. HMAC-SHA1 and AES were used for authentication and encryption/decryption of partial bitstream but any arbitrary algorithm can be used as well. Also the HWICAP module, used for reconfiguration, is controlled through software (ICAP API) which facilitates reconfiguration. EDK automatically creates the memory map of the systems.

4.3 Hardware Architecture

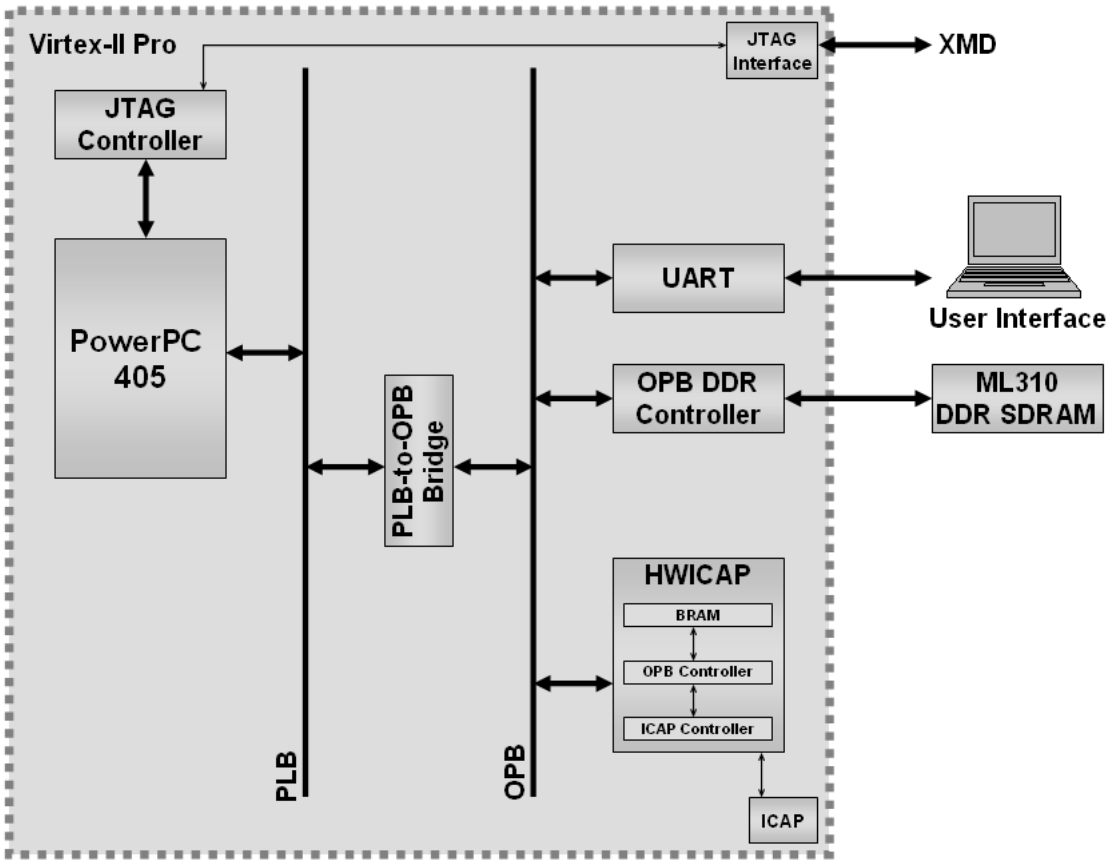
Figure 4.1 and 4.2 show the hardware components of the constructed self-reconfiguring platforms targeting both 32-bit embedded PowerPC hard processor core and MicroBlaze soft processor core. The systems were implemented in a XC2VP30-FF896-6 Virtex-II Pro FPGA device on the ML310 Evaluation Board with minimal footprint. Instruction cache and data cache options are disabled for the microprocessors and both systems run at 100MHz.

4.3.1 Processor Cores and Buses

The embedded PPC405 hard processor core in Virtex-II Pro is an implementation of the embedded PowerPC 405D5. It is a 32-bit Harvard architecture processor and contains elements such as: a five-stage pipeline, virtual-memory management unit, timers, and separate instruction and data cache units.

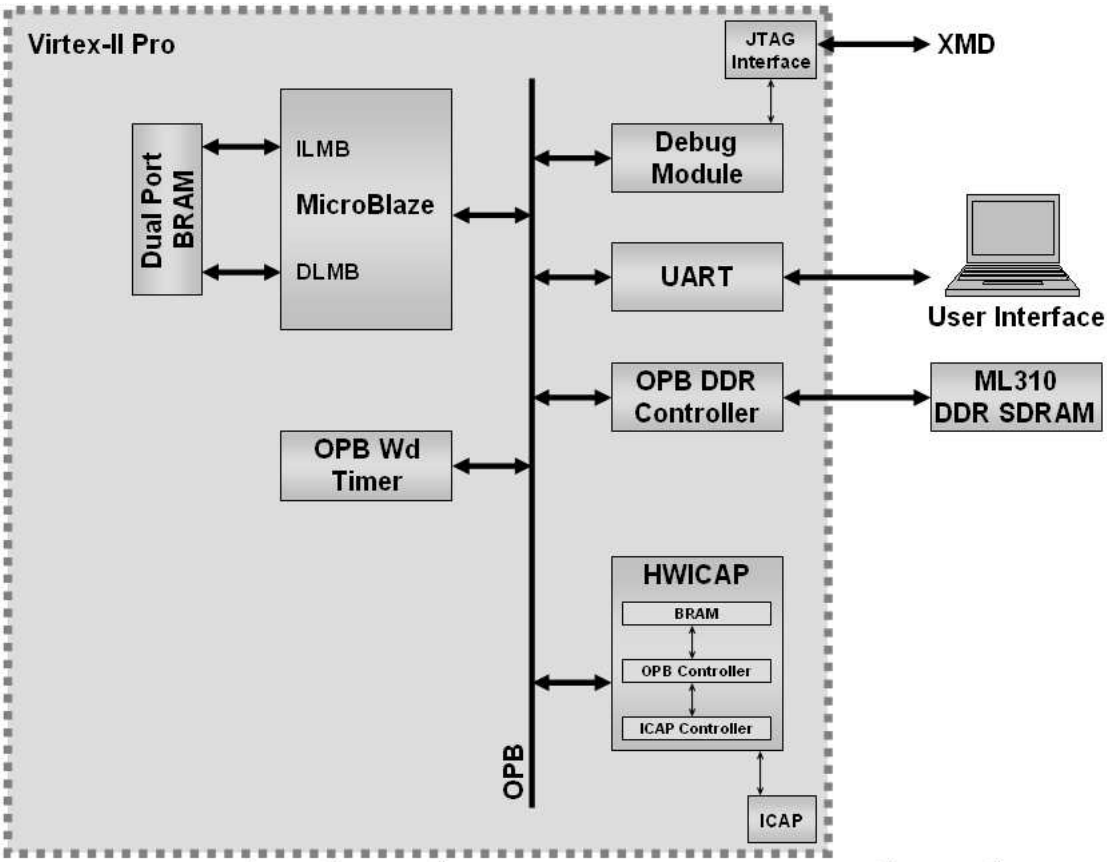
The MicroBlaze soft processor core is a true 32-bit processor that supports 32-bit bus widths. The core is a RISC-based engine with a 32-bit LUT RAM-based register file with separate instructions for data and memory access. It supports both on-chip Block-RAM and/or external memory through a LMB (Local Memory Bus). MicroBlaze uses a three-stage pipeline and separate instruction and data cache units.

Both embedded PowerPC and MicroBlaze processor cores communicate with peripherals through one or more of the IBM CoreConnect buses [16] which enables compliant IP cores to integrate with embedded processor cores. The CoreConnect bus architecture provides three buses for interconnection of hard/soft IP cores. The key features of CoreConnect bus architecture are:



PLB = Processor Local Bus, OPB = On-chip Peripheral Bus.
ICAP = Internal Configuration Access Port, HWICAP = Hardware ICAP,
XMD = Xilinx Microprocessor Debugger.

Figure 4.1: PowerPC System



ILMB = Instruction-side Local Memory Bus, DLMB = Data-side Local Memory Bus, OPB Wd Timer = On-chip Peripheral Bus Watchdog Timer, ICAP = Internal Configuration Access Port, HWICAP = Hardware ICAP, XMD = Xilinx Microprocessor Debugger.

Figure 4.2: MicroBlaze System

- *Processor Local Bus (PLB)* – Used by cache units of PPC405 core to access high-speed system resources and high-performance peripherals. It has a synchronous architecture with separate busses for address (32-bit) and data (64-bit), and independent data paths for read/write.

- *On-chip Peripheral Bus (OPB)* – This bus has 32-bit address and data lines and supports a greater number of devices which results in decoupling low-bandwidth peripherals from the PLB and reducing the PLB traffic.

- *Device Control Register (DCR) Bus* – Used by PPC405 core to initialize and control peripherals devices that reside on the same FPGA chip. It is a 32-bit bus directly accessible by PowerPC general purpose registers. DCR bus reduces PLB traffic and improves system integrity [9].

Both implemented systems require the OPB bus to instantiate the HWICAP module for reconfiguration since the current implementation of this module only connects to OPB. PowerPC only has the PLB bus interface and therefore OPB devices cannot directly connect to the processor. CoreConnect bus structure facilitates hierarchical bus design by allowing various types of bridges (OPB2PLB and PLB2OPB). These bridges connect multiple segments of the bus. In PowerPC system, processor and peripherals communicate over the OPB connected to the PLB through PLB2OPB bridge.

Both Xilinx PLB and OPB modules provide efficient arbitration. In a processor-based system, bus masters initiate a bus transaction and bus slaves devices can only respond to master request. When more than one master device is present, an arbiter is required to guide the master devices when they can drive the buses.

The MicroBlaze system is configured with OPB bus and two LMBs. The LMB is a fast and efficient local bus with separate read and write data buses. It requires no arbiter (single master bus) and connects MicroBlaze instruction and data ports to high-speed peripherals, primarily BRAMs. The LMB provides single-cycle access to on-chip dual-port block RAM and operates at 125 MHz.

4.3.2 HWICAP (Hardware Internal Configuration Access Port) Module

The HWICAP module [17] is used for reconfiguration. It enables the microprocessors to read and write the FPGA configuration memory as well as loading partial bitstreams from system memory through ICAP. The HWICAP core shown in figure 4.3 consists of OPB controller, ICAP controller, and a BRAM. It uses the BRAM on OPB bus as a configuration cache and has the capability to transfer the partial bitstream from local memory via the OPB to ICAP. The partial bitstream is transferred frame by frame to this BRAM and then to ICAP. The HWICAP ICAP controller connects to the ICAP block located in the lower right hand corner of the FPGA. ICAP interface operates at the clock rate of OPB bus.

4.3.3 Other Peripherals

Memory Devices

In MicroBlaze system both ILMB and DLMB (Instruction- and Data-side Local Memory Buses) are connected to the 8KB of dual-port BRAM using different ports of the BRAM. In PowerPC system 8KB of on-chip BRAM is connected to the OPB bus. This memory is used for bootloop storage in both systems. Bootloop consists of a

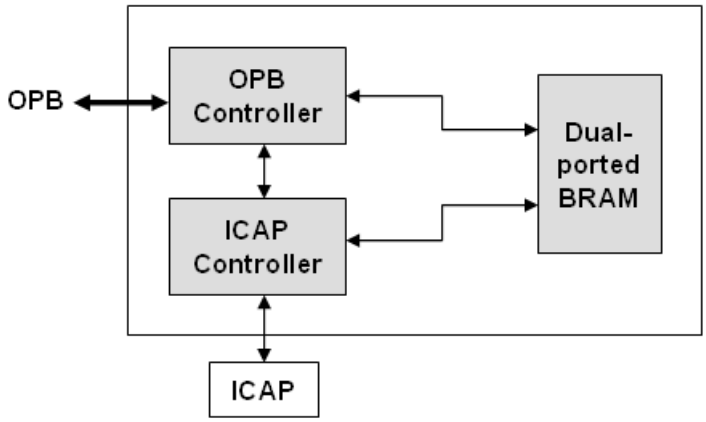


Figure 4.3: Block Diagram of HWICAP Module

simple branch instruction, and is located at the processors boot location. The purpose of a bootloop application is to keep the processor in a defined state until the actual application can be downloaded and run. XPS contains a predefined bootloop application for both PowerPC and MicroBlaze processor cores.

DDR SDRAM on the ML310 Evaluation Board, connected to an External Memory Controller on the processor OPB bus was selected as the external memory for storage of the encrypted partial bitstreams. This DDR controller module performs device initialization and auto-refresh cycles of up to four DDR memory banks.

Clock/reset Distribution

Two Digital Clock Manager (DCM) modules are used for providing clocks. A 100 MHz input reference clock is used to generate the main 100 MHz PLB, OPB, and LMB clocks. The CLK90 output of the DCM produces a 100 MHz clock that is phase shifted by 90 degrees for use by the DDR SDRAM controller.

The Xilinx Processor System Reset Module provides three types of reset supported by the PPC405:

- *Core Reset* – Only affects the processor
- *Chip Reset* – Clears all the logic on the FPGA
- *System Reset* – Resets the entire system including the FPGA and external devices connected to the FPGA

The reset interface in the processor block enables the PPC405 core to recognize resets generated by three user reset input pins. MicroBlaze does not support different levels of reset.

CPU Debug via JTAG

CPU JTAG chain is combined with the FPGA's main JTAG chain to download bitstreams for FPGA programming as well as CPU software debugging. The debug interface of Processor Block provides access to resources internal to the core and assist in software development. Sharing the same JTAG chain simplifies the number of cables needed since only a single JTAG cable (like the Xilinx Parallel IV Cable) is needed.

JTAG port was used for both transferring the partial bitstream to DDR memory and debugging. PowerPC system requires a JTAG controller that allows the PowerPC to connect to the JTAG chain of the FPGA instantiating a JTAGPPC primitive and directly connecting it to both PowerPC CPUs in the chip. MicroBlaze system requires a Microprocessor Debug Module on the processor OPB bus for JTAG-based debugging. This module can also be used with PowerPC 405 processors.

Additional Features

RS232 serial channel on the ML310 Evaluation Board, connected to a UART peripheral on the processor OPB bus used for stdin and stdout.

In MicroBlaze system a free-running Timebase and Watchdog Timer peripheral on the processor OPB bus was used to for timing. PowerPC provides a 64-bit timebase counter inside the processor that works with the system clock and thus, no extra component is needed.

UART and additional features are not the essential parts of the self-reconfiguring systems but provide ease of use for user application. Detail description of the IP cores used in the designs is provided in Processor IP Reference Guide included with the EDK [17].

4.4 Software Architecture

4.4.1 Overview

EDK automatically generates the memory map of the hardware platform as well as assigning default drivers to the processors and each of the peripherals. The program running on the processor core uses some basic standard C libraries and device drivers since there is no operating system between the software and the hardware platform.

The software performs the following tasks:

- *Authentication* – Verifying the signed partial bitstream with the stored MAC value.
- *Decryption* – Decrypting the encrypted partial bitstream using the stored key.

- *Configuration* – Partially reconfiguring the other active system on FPGA using the decrypted partial bitstream.

The source code for the program is written in C. It is compiled and linked to generate executable files in the ELF (Executable and Link Format) format. The program is stored in the external DDR memory. Using the Xilinx Microprocessor Debugger (XMD), the signed and encrypted partial bitstream is also stored in DDR memory at an address range not used by the program.

The following sections describe different parts of the software in more detail.

4.4.2 AES and HMAC-SHA1 Algorithms

To perform decryption and authentication, the program uses AES and HMAC-SHA1 functions implemented by Dr. Brian Gladman [18]. The source code is available in C/C++ for anyone to use under an open source license on author's web site. The functions are portable with no OS or library dependency. The source codes were imported as a user library to EDK. Information on how the user can add libraries and customize peripherals and associated drivers can be found at [13]. After adding the source codes as libraries, they can be simply selected for project using Software Platform Setting.

The AES code is supplied in the files aes.h, aesopt.h, aescrypt.c, aeskey.c, and aestab.c. The file to be encrypted or decrypted is split up into 16 byte blocks (the last can be a partial block) and the resulting block number is used as the encryption/decryption nonce for CTR mode. The block size used for encryption and decryption must be the same. The lengths of the encrypted file and the non-encrypted file are also the same.

Authentication ensures that the contents of an encrypted file have not been changed or tampered with after encryption. It is useful because it blocks the attackers whose goal is not only capturing the bitstream but also damaging the FPGA itself. It is even more essential when using CTR mode encryption because this mode is vulnerable to several trivial attacks without authentication. The message authentication algorithm HMAC-SHA1 is supplied in hmac.h and hmac.c and the SHA1 hash code is provided in the files sha1.h, sha1.c. It is only necessary to include aes.h and hmac.h in the program. The MAC value of the encrypted partial bitstream and the secret key are also stored in the application.

HMAC-SHA1

A keyed Hash Message Authentication Code, or HMAC, is a type of message authentication code (MAC) calculated using an iterative cryptographic hash function, such as SHA-1, in combination with a secret key. It verifies both the data integrity and the authenticity of a message. It also provides compression since any arbitrary length input result in a fixed length output. HMAC is defined as:

$$HMAC(m) = \ h\ (K \oplus ipad\ \ ||\ \ h\ (K \oplus opad\ ||\ m)), \tag{4.1}$$

where h is the hash function, K is a secret key, m is the input message, and *ipad/opad* are constant padding strings of the length of the message block size in the hash function h.

The implementation of HMAC used in the application provides a subroutine to compute the MAC value of the encrypted partial bitstream:

```
void hmac_sha1(const unsigned char key[], unsigned int key_len, const
```

`unsigned char data[], unsigned int data_len, unsigned char mac[], unsigned int mac_len);`

This subroutine calls the following functions:

- `hmac_sha1_begin()` – Initializes the HMAC context to zero.
- `hmac_sha1_key()` – Inputs the HMAC key.
- `hmac_sha1_data()` – Inputs the HMAC data.
- `hmac_sha1_end()` – Computes and outputs the MAC value.

The following parameters are passed to this subroutine:

- *key* – secret key stored as a constant array in the program
- *key_len* – the size of key in bytes (16 bytes)
- *data* – a pointer to the partial bitstream in external memory
- *data_len* – size of the partial bitstream in bytes
- *mac* – storage array for MAC value
- *mac_len* – size of the MAC value output (20 bytes)

The application then compares the generated MAC value by this subroutine with the stored MAC value in the program. If both values are equal authentication is successful and as a result the decryption phase can be started.

AES

Advanced Encryption Standard (AES) algorithm, supported by NIST as an official encryption standard, is a block cipher that uses 128, 192, or 256 bits keys to encrypt or decrypt blocks of 128 bits of data at a time. NIST selected Rijndael cipher developed by two Belgian cryptographers as the AES algorithm.

Rijndael key and block sizes can be sequences containing 128, 160, 192, 224 or 256 bits. Rijndael operates on a two dimensional array of bytes called the state. For encryption, each round of Rijndael (except for the last round) consists of four stages:

- *The SubBytes Transformation* – Acts on every byte of the state to produce a new byte using an S-box substitution table.
- *The ShiftRows Transformation* – Each row of the state is shifted cyclically a certain number of steps.
- *The MixColumns Transformation* – Acts independently on every column of the state and combines the four bytes in each column using a linear transformation
- *The AddRoundKey Transformation* – Each byte of the state is combined with the round key.

Decryption can be implemented in the same form as the encryption cipher by applying a series of transformations. This is possible because the order of some operations in decryption can be changed without affecting the final result.

In software, Rijndael can be implemented more efficiently on processors with 32-bit words. The optimized implementation converts transformations of the entire round into look-up tables. Different tables are required for the last round.

The implementation of AES used in the application uses the following subroutines and functions to perform decryption:

- `gen_tabs()` – generates the tables
- `f_dec_key128()` – initializes the key schedule from the user supplied key
- `aes_ecb_decrypt()` – decrypts a 16-bytes block

A structure for the key scheduling is allocated with the maximum 60 word array. `f_dec_key128()` takes as parameters a pointer to a variable with this type and the secret key stored as a constant array in the program. `gen_tabs()` routine must be called before first use with no parameter. The number of 16-bytes blocks in the partial bitstream is set before starting the decryption. Then `aes_ecb_decrypt()` routine is called inside a loop to decrypt all the 16-bytes blocks of the partial bitstream:

```
aes_ecb_decrypt(const unsigned char *in, unsigned char *out, int no_block,
const aes_decrypt_ctx cx[1])
```

The following parameters are passed to this function at each iteration of the loop:

- *in* – secret key stored as a constant array in the program
- *out* – a pointer to the partial bitstream in external memory (increased by 16 at each iteration)
- *no_block* – a pointer to the storage array for the decrypted partial bitstream in external memory (increased by 16 at each iteration)
- *cx* – constant '1'

4.4.3 ICAP API

The ICAP API [19] was used for transferring the data between the external configuration memory and OPB BRAM configuration cache. The ICAP API defines methods for accessing the configuration logic through ICAP port. For using HWICAP API, xhwicap.h should be included in the application. A structure (XHwIcap) holds the driver instance data. It is required to allocated a variable of this type for HWICAP module in the system. A pointer to a variable of this type is then passed to the driver API functions.

Before using the HWICAP module, one must initialize it. `XHwIcap_Initialize()` should be called for initialization. The function requires the following parameters: a pointer to the XHwIcap instance, base address of the instance, user defined ID for the instance, and IDCODE of the FPGA device. IDCODE can be read from the device by using the XHI_READ_DEVICEID_FROM_ICAP constant. After successful initialization the following function loads a partial bitstream from external memory.

`XHwIcap_SetConfiguration(XHwIcap *InstancePtr, Xuint32 *Data, Xuint32 Size)`

The following parameters are passed to this function in the application:

- *InstancePtr* – a pointer to the XHwIcap instance to be worked on.
- *Data* – a pointer to the storage array for the decrypted partial bitstream in external memory
- *Size* – the size of the partial bitstream in 32-bit words.

Chapter 5: Experiment Methodology

5.1 Overview

To perform an experiment using the self-reconfiguring systems described in the previous chapter, the first step was generating a partial bitstream. This requires implementing a partially reconfigurable design consisting of fixed and reconfigurable modules.

The reconfigurable design in our experiments consists of two separate processor based systems:

- *Static System* – The PowerPC/MicroBlaze self-reconfiguring system was considered to be the static/fixed module of the design.
- *Reconfigurable System* – An additional PowerPC/MicroBlaze system was implemented in the form of a microcontroller as the target of partial reconfiguration. This system generates a pattern on the ML310 LEDs that are connected to the reconfigurable system general-purpose IO (GPIO) ports.

The considered scenario for the experiment is as follows. The self-reconfiguring systems read an authenticated and encrypted partial bitstream stored in an external memory, authenticate and decrypt it, and send it to ICAP to change the reconfigurable system. The secret keys is stored in the program running on the self-reconfiguring platforms even though other cases such as providing the key as a password are also possible.

The difference-based and module-based methods were both employed to create the reconfigurable design but only the former resulted in a working partial bitstream. The complete detail of steps taken for each flow is described in the following sections along with encountered problems and limitations.

5.2 Difference-Based Reconfigurable Design

An additional MicroBlaze system was implemented in the form of a microcontroller as the target of partial reconfiguration. Using EDK this system was combined separately with each of the self-reconfiguring systems. This can be done either by directly changing the MHS file or using the Add/Edit cores dialog box. Figure 5.1 shows a block diagram of this system and the static system. The reconfigurable system only includes:

- MicroBlaze 32-bit soft processor core
- 2 x LMB Bus (ILMB and DLMB)
  - 2 x LMB_BRAM_IF_CNTLR
  - LMB BRAM (8 KB)
- OPB BUS
  - OPB_GPIO connected to the 8-bit LED display on the ML310 board.

The partial bitstream changes the BRAM contents where the program running on the MicroBlaze system had been stored. Since the application running on this system was generating a pattern on LEDs, partial reconfiguration would result a different pattern to appear on LEDs.

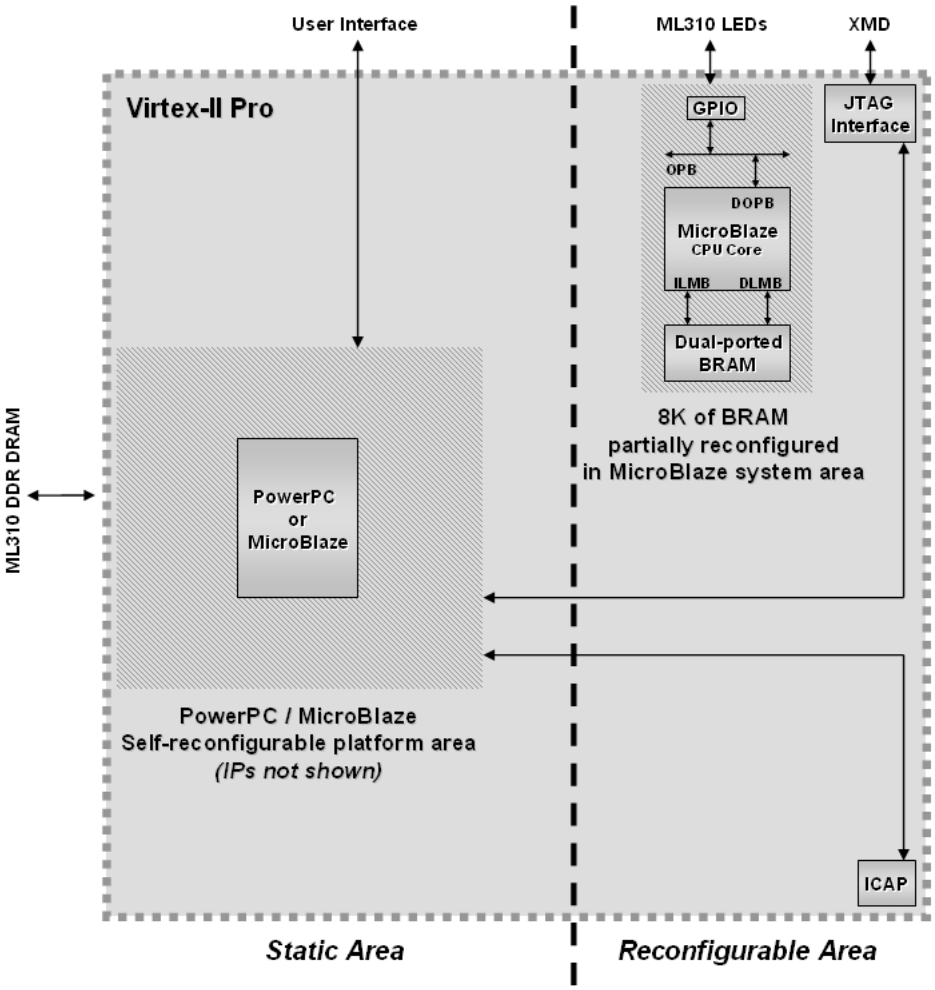


Figure 5.1: Simplified Layout of the Difference-Based Experiment Design

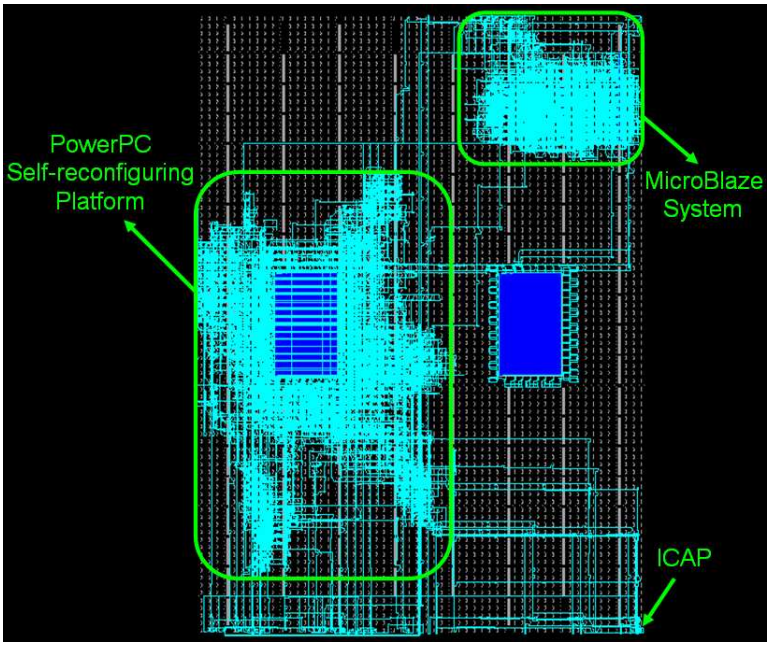


Figure 5.2: FPGA Editor View of the Difference-Based Design

Figure 5.2 shows the FPGA Editor view of the design. The implemented design file with '`.ncd`' extension is opened with FPGA Editor tool to modify the BRAM contents of the reconfigurable system. This file should be immediately saved under a different name so that the original design is not lost. Also file property should be changed to Read Write in the File menu. Using Block Editor Mode each BRAM is selected and its initial values is modified with the values obtained in system initialization HDL file. A simpler way of loading the contents of BRAMs is opening the software executable file with '`.elf`' extension along with BRAM memory map file with '`.bmm`' extension at the time of opening the implemented design file in the FPGA Editor.

This way the modified design file must be used with the initial bitstream for creating a difference-based partial bitstream. The initial bitstream of the design contained the original BRAM contents. The partial bitstream was created with BitGen program

using the -r switch. BitGen set with this switch produced a bitstream that contained only the differences between the modified design file and the initial bit file. The -g ActiveReconfig:Yes switch is also required for dynamic partial reconfiguration. The generated bitstream (14 KB) was much smaller than the initial bitstream (1.38 MB). BitGen ensures that the partial bitstreams would be as small as possible by producing them with "Multi Frame Writes." Using this feature BitGen can write multiple frames at once, and as a result it also generates the partial bitstreams with different sizes depending on the design of the module.

After encrypting and signing the partial bitstream, the initial FPGA bitstream was downloaded into the JTAG port of the FPGA on ML310 Evaluation Board. Then Xilinx Microprocessor Debugger was used to download the partial bitstream from the host machine (connected to the board) to an address range not used by the program in DDR memory on the board.

The next step was running the program on the self-reconfiguring system to perform the required tasks. It successfully authenticated the signed partial bitstream with the stored MAC value; decryption phase would not start if the generated MAC were different than the stored MAC in program. The program then decrypted the encrypted partial bitstream using the stored key, and dynamically partially reconfigured the other active system on the FPGA. The experiment was judged to be successful when the new pattern was displayed on the LEDs of the board. It was verifying that the new application was correctly replaced the initial program stored in the internal BRAMs of the MicroBlaze system.

Chapter 6 presents the timing results obtained for execution of each phase of the application along with the device resource utilization summary.

5.3 Module-Based Reconfigurable Design

5.3.1 Design Entry and Synthesis

Figure 5.2 shows the implementation of the top-level design in this experience with the instantiation of systems and bus macros. Similar to the previous flow the top-level design consists of the following modules:

- *Static System* – The PowerPC self-reconfiguring platform described in the previous chapter.
- *Reconfigurable System* – Contains the other PowerPC processor available on the chip, PLB and OPB buses, 8K of BRAM connected to OPB, and OPB GPIO connected to the 8-bit LED display on the ML310 board.
- *ICAP and JTAG Wrapper Module* – Instantiates the ICAP and JTAG primitive components.

Using PlatGen both static and reconfigurable systems were generated as sub-modules in EDK with some modified IP cores and additional external ports and connections required for the bus macros. The ICAP and JTAG wrapper module was synthesized and instantiated separately in the top-level. All the guidelines provided in section 3.2.2 were considered in the design entry and synthesis of the top-level design and each module.

Each module separately provided local constants for the enable/disable ports of the bus macros by adding external ports for VCC and GND. It is required since using constant values in the top-level design might cause sharing of these signals which is prohibited by the rules of the module-based flow.

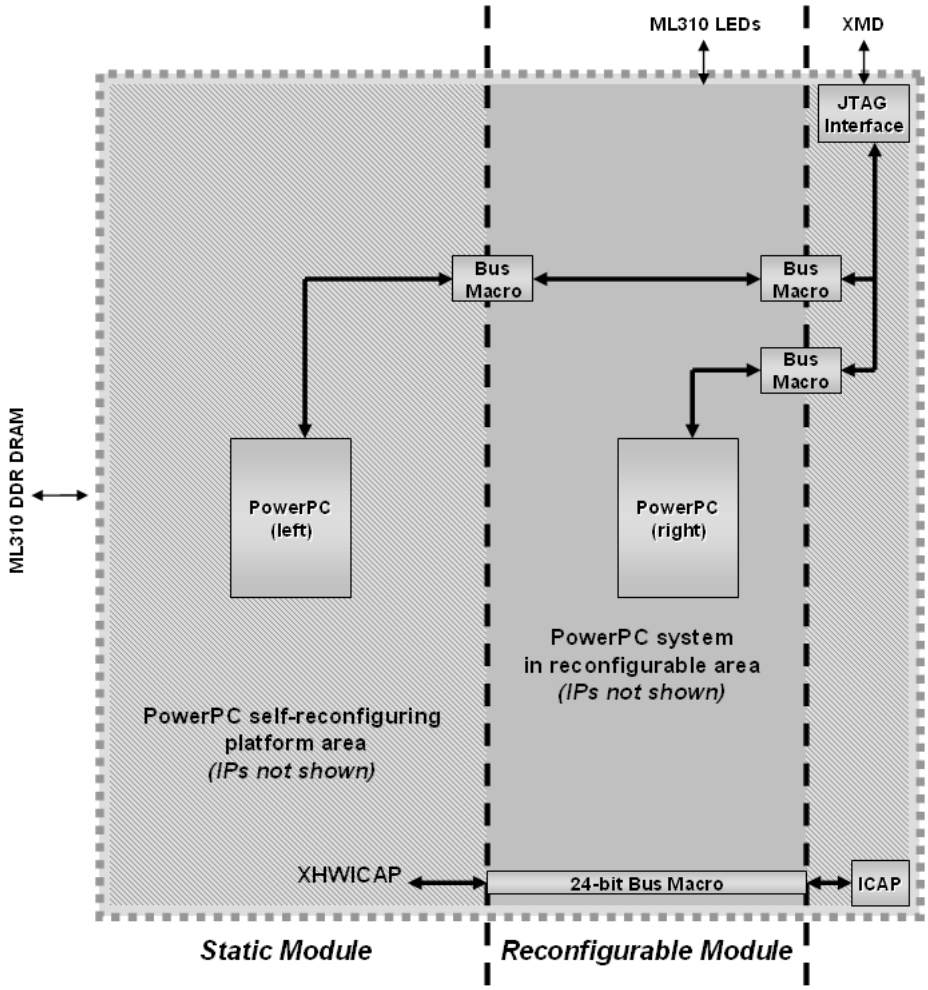


Figure 5.3: Simplified Layout of the Module-Based Experiment Design

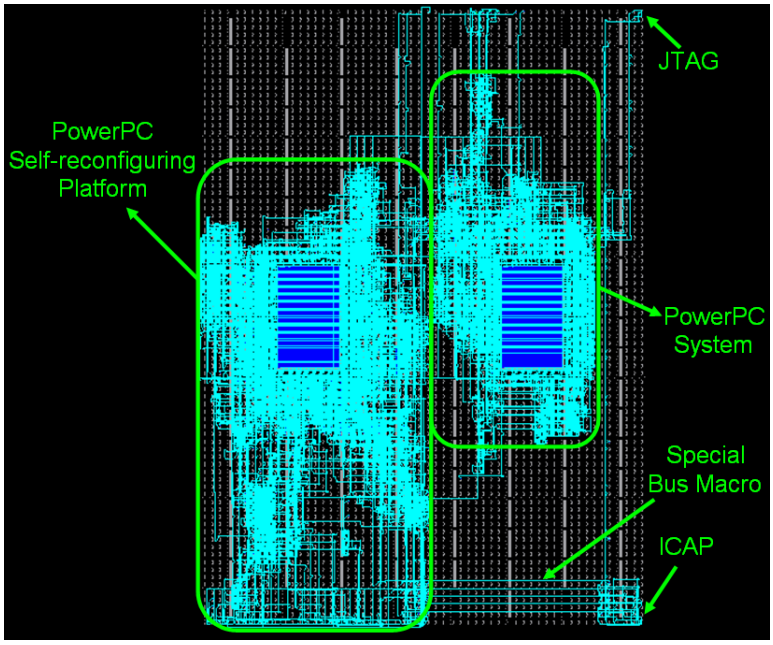


Figure 5.4: FPGA Editor View of the Module-Based Design

Clock signal coming from a global clock buffer was the only signal shared by the modules. A global clock buffer distributes high fan-out clock signals throughout a device. In Virtex-II Pro clock buffers are multiplexed clock buffers (BUFGMUX) that can select between two input clocks. This buffer needed GND signal for its select input. The constant signal was provided by the static module because the BUFGMUX was in the boundary of the static module.

5.3.2 Bus Macros

The self-reconfiguring system used many pins of the ML310 board scattered all around the chip. For example all the pins for DDR memory controller were at the bottom and left part of the device while the ICAP and JTAG components were at the right part. On the other hand module-based flow requires a module to use only the resources

in its placement area and therefore it was not possible to assign a boundary for this system that encompasses all the resources it needs. Bus macros were needed for all the signals that passed through the reconfigurable module boundary even though there was no inter-communication between the static and reconfigurable system.

To minimize the number of these signal, the static module was placed in the left part of the device while the reconfigurable module is at the right part. This way only signals for ICAP and JTAG must pass through the reconfigurable part. Both of these primitive components were being instantiated in the HDL code of HWICAP and JTAGPPC processor IP cores used in the self-reconfiguring system. Therefore the code for HWICAP IP core was modified in a way that ICAP primitive component would not instantiate in the static module and instead all the inputs and outputs of this component were treated as external ports. JTAGPPC was also removed from the static module and instead FPGA JTAG chain with two PowerPC processors was manually created by investigating the HDL code of JTAGPPC IP core and adding additional external ports for each module. All the required pins of the two PowerPCs were connected to external ports so that they could be connected to bus macros and eventually to JTAG primitive at the right part of the device.

As it was required to keep the signals connected to ICAP primitive active during the reconfiguration it was necessary to use a custom bus macro that span the width of the reconfigurable module. This bus will stay intact during reconfiguration since it is exactly placed the same way before reconfiguration. The bus macro provided by Xilinx was not useful as it only spans four columns and is designed for communications of modules adjacent to each other. It should be noted that reconfiguration is glitchless for unchanged resources. This custom bus macro was created using FPGA Editor.

Additional bus macros were also used for the static module UART pins in the reconfigurable area and providing the reset signal in the static area to reconfigurable module.

5.3.3 Implementation

Initial Budgeting Using UCF constraint file, an ASCII file specifying constraints on the logical design, three area groups were created in accordance with the guidelines in section 3.2.2:

1. AREA_GROUP "AG_U1" RANGE = SLICE_X0Y159 : SLICE_X47Y0
2. AREA_GROUP "AG_U2" RANGE = SLICE_X48Y159 : SLICE_X83Y0
3. AREA_GROUP "AG_U3" RANGE = SLICE_X84Y159 : SLICE_X91Y0

Instances of the static, reconfigurable, and ICAP-JTAG wrapper modules were assigned to the above area groups respectively. TBUFs, BRAMs, and MULTs were also included in the ranges of the area groups. Also, all system level, FPGA pins, and bus macro location constraints were assigned.

Active Module Implementation Module specific constraints such as DCM and PPC405 location constraints were added to the constraint file created in the previous phase.

To implement each module, the top-level and module netlists in NGC format and the constraint file were used. Each active module was separately expanded into the top-level design. It then mapped, placed, and routed. The implemented designs were published as Physically Implemented Module (PIM) to a different directory to be used in the next phase. The following commands were used in this phase:

1. *ngdbuild -p xc2vp30ff896-6 -uc <constraint file>.ucf -modular module -active <module name> <top-level file>.ngc*

2. *map -pr b <top-level file>.ngd -o <top-level file>_map.ncd <top-level file>.pcf*

3. *par -w <top-level file>_map.ncd <top-level file>.ncd <top-level file>.pcf*

4. *pimcreate -ncd <top-level file>.ncd -ngm <top-level file>_map.ngm ...\Pims*

Final Assembly This phase constructed a complete design using the top-level design files created during the initial budgeting phase, the top-level constraints file, and the implemented modules published to the ...\Pims directory in the previous phase. This integration was done in the ...\Assemble subdirectory corresponding to each top-level design. The final bitstream for the complete design was generated in this phase after running the following commands:

1. *ngdbuild -p xc2vp30ff896-6 -uc <constraint file>.ucf -u -modular assemble -pimpath ...\...\Pims <top-level file>.ngc*

2. *map -pr b <top-level file>.ngd -o <top-level file>_map.ncd <top-level file>.pcf*

3. *par -w <top-level file>_map.ncd <top-level file>.ncd <top-level file>.pcf*

Up to this point we had one complete design. Any variation of the reconfigurable module should be created and implemented following the same procedure described in the previous sections. After that the partial bitstreams generated for each variation of the reconfigurable modules can change the configuration from one design to the next one.

Two alternate processor systems were attempted to integrate with one of the self-reconfiguring systems. The first system was a MicroBlaze system with the same functionality as the reconfigurable system implemented before and the second system was a similar PowerPC system except that it only used the PLB bus and GPIO. Unfortunately, these alternate reconfigurable systems faced different problems, discussed in the following section, during the implementation process. Hence we were not able to experiment with the partial bitstreams generated by this flow.

5.3.4 Problems

Most of the minor problems encountered during the design entry, synthesis, and implementation phases were usually resolved by reviewing the guidelines for each phase or tweaking some options in the tools. Xilinx support provides some answer records that might be helpful to resolve the problems.

Major problems were usually encountered during the final assembly phase of the implementation flow for the systems. Different kinds of "FATAL_ERROR" and "INTERNAL_ERROR" messages usually made PAR too to fail in the final assembly stage. Mostly, the problem is either unknown or no work around is available. Some of the encountered problems are listed below.

Error: PAR - FATAL_ERROR:Route:basrtareacst.c:792:1.6.12.3...

Solution: Use of non-rectangular area group ranges

Error: MAP - Pack:625 - The dual data rate register "REG0" failed to combine ...

Solution: Not using the -u command line option

Error: GLOBAL_LOGIC - Creating and/or using VCCs and GNDs

Solution: MODE=RECONFIG option for AREA_GROUP or LUT instantiation

Error: PAR, BUS MACRO - PAR fails after Phase 1.1 and reports no ...

Solution: Service pack update

Error: PAR - Guided placement fails, reporting:
"INTERNAL_ERROR:Place:basplrpmsupp.c:484:1.13.2.1 - PARTIALLY PLACED MACRO ENCOUNTERED!!"

Solution: Currently under investigation by Xilinx, applying an RLOC_ORIGIN constraint to any affected macro might help

Using online forums and list archives [20] may also provide some additional information about the problems.

As it seems partial reconfiguration using module based flow works better for simple designs with few modules. The tools become problematic as the design gets larger and more complex. If possible, it is best to work with Xilinx support on a specific design case to work around the problems and save valuable time. Also it is recommended to study the pin assignment on a board for partial reconfiguration purpose before purchasing it since most of the boards in the market were not designed for this purpose.

5.4 Evaluation of Partial Reconfiguration Flows

Table 5.1 and 5.2 provide the evaluation summary for module-based and difference-based flows.

Table 5.1: DIFFERENCE-BASED FLOW EVALUATION

Level of required effort	Medium depending on the changes made and level of acquaintance with the tool
Level of support of existing tools	Acceptable with occasional errors and problems
Practical limitations	Not recommended if routing changes is desired
Benefits	Small partial bitstreams (Multiple-frame Write)

Table 5.2: MODULE-BASED FLOW EVALUATION

Level of required effort	High; needs more than average acquaintance with the tool
Level of support of existing tools	Problematic with frequent errors especially for complex designs
Practical limitations	Requires: - A full design for initial reconfiguration - Special consideration for inter-module communications - Different constraints for modules
Benefits	Automation and bounded routing

5.5 Security Analysis

One of the main advantages of using the self-reconfiguring systems is the increase of flexibility. The designer is able to partition the application according to the necessary security level and choose the suitable algorithms for the authentication and decryption. Moreover these algorithms can be upgraded to take advantage of the latest improvements of the security field without any change in the implemented partially reconfigurable design.

The following considerations should be taken into account to improve the security of the scheme:

- *Partial Bitstream Storage* – Storing the partial bitstream in internal memory prevents the interception of the bitstream after decryption. The program running on the processor core should be modified in a way that either one block of the partial bitstream is decrypted and sent to ICAP at a time or there is enough internal memory to store the whole decrypted partial bitstream in the FPGA.

- *Key Storage* – Storing the key in a battery-powered storage or providing the key by an user entered password are among the options that can be used instead of storing the key in the program.

Chapter 6: Results

6.1 Timing Measurements

In Table 6.1 the timing result of each phase for both self-reconfiguring systems in difference-based experiment is provided. For each phase of the process (authentication, decryption, and configuration) 10 measurements were done by obtaining the number of clock cycles required for each processor to execute the functions. For PowerPC system no extra component was needed since a time-base register inside the processor is available that works with the system clock. For MicroBlaze system a watch-dog timer on OPB was used that contains a time-base register. For both systems, standard deviation from the mean value at each phase along with the percentage error is also shown in Table 6.1.

Table 6.2 summarizes the comparison of the results for the average values and throughput. The average values of the obtained results show that PowerPC system performed faster in both authentication and decryption phases of the application. Consequently it has higher throughput in these two phases with the ratios shown in the table. Even though both systems were running at 100 MHz, the better performance of the PowerPC system could be due to the fact that its instruction set executes most of the instructions in a single cycle and is more efficient than MicroBlaze. On the other hand, MicroBlaze system gives a better performance working with the HWICAP module and therefore it achieves a higher throughput for configuration. The

Table 6.1: TIMING RESULTS FOR EACH PHASE (CLOCK CYCLES)

PowerPC System

Measurement	Authentication	Decryption	Configuration
1	13,862,435	20,838,769	5,630,038
2	13,862,591	20,838,876	5,631,061
3	13,862,486	20,838,769	5,630,038
4	13,862,435	20,838,769	5,630,038
5	13,862,500	20,838,769	5,631,037
6	13,862,575	20,838,776	5,630,038
7	13,862,591	20,838,876	5,628,993
8	13,862,575	20,838,776	5,630,038
9	13,862,591	20,838,879	5,628,993
10	13,862,486	20,838,769	5,631,037
Std. Dev.	65	51	756
Mean	13,862,527	20,838,803	5,630,131
% Error	0.05%	0.02%	1.34%

MicroBlaze System

Measurement	Authentication	Decryption	Configuration
1	77,649,436	147,201,543	3,175,996
2	77,649,453	147,201,601	3,175,964
3	77,649,510	147,201,675	3,175,420
4	77,649,416	147,201,543	3,175,996
5	77,649,510	147,201,675	3,175,943
6	77,649,349	147,201,675	3,175,996
7	77,649,597	147,201,639	3,175,996
8	77,649,597	147,201,675	3,175,952
9	77,649,515	147,201,451	3,176,008
10	77,648,899	147,201,639	3,175,996
Std. Dev.	201	77	179
Mean	77,649,428	147,201,612	3,175,927
% Error	0.03%	0.01%	0.56%

Table 6.2: COMPARISON OF THE TIMING RESULTS FOR EACH PHASE

Measurement	System	Authentication	Decryption	Configuration
Ave. Time (ms)	PowerPC	139	208	56
	MicroBlaze	776	1472	32
Through-put (KB/s)	PowerPC	102	68	251
	MicroBlaze	18	10	444
Ratio	PPC / MB	5.6	7.0	0.5

Timing Based on Units of Operation

System	Clock Cycles / Byte	Clock Cycles / 16 Bytes Block	Clock Cycles / 4 Bytes Word
PowerPC	982	23,627	1,596
MicroBlaze	5,502	166,895	900

reason might be the presence of the extra PLB bus and PLB to OPB Bridge in the PowerPC system. Since HWICAP module is a slave on the OPB bus the processor should transfer the frames of the bitstream from the DDR to the HWICAP BRAM and therefore an extra bus may actually increase the time of this transfer. Thus, DMA data transfer is desirable to increase the performance of HWICAP in any system.

Table 6.2 also provides the time based on the unit of operation for each phase. Authentication algorithm works on bytes with a total number of 14112 bytes in the partial bitstream. Decryption works on blocks of 16 bytes in CTR mode. There were 882 blocks in the partial bitstream. Also, 32-bits words are sent to ICAP for reconfiguration with the total number of 3528.

Table 6.3: DEVICE UTILIZATION SUMMARY

Device Resources	Number of Resources				
	Used by PowerPC		Used by MicroBlaze		Available in Device
	Total	Percentage	Total	Percentage	
SLICEs	1334	9	1706	12	13696
RAMB16s	5	3	5	3	136
MULT18X18s	0	0	3	2	136
BUFGMUXs	7	43	8	50	16
DCMs	2	25	2	25	8
JTAGPPCs	1	100	1	100	1
ICAPs	1	100	1	100	1
PPC405s	1	50	0	0	2

6.2 Device Utilization Summary

Systems were designed with only the required components. It should be noted that the Xilinx MicroBlaze soft processor uses 950 logic cells (475 Slices) in the Virtex-II Pro device but PowerPC cores are part of the FPGA fabric with no resource usage even though hard core processors in the FPGA fabric reduce the available area for logic in general. Table 6.3 provides the device utilization summary for both systems.

The resource utilization is only for the self-reconfiguring platforms in the design and not the additional MicroBlaze system under reconfiguration. The device utilization is close for both systems. The PowerPC system used lesser amount of resources even though it required the use of extra PLB bus and PLB to OPB Bridge but it should be considered that the resource usage for the MicroBlaze system includes the soft processor as well.

Table 6.4 provides the contribution of different IP cores. The required IPs for both systems are listed on the top section of the table followed by the necessary

Table 6.4: RESOURCE USAGE OF IP CORES

System Component	Resources Used					
	Slices		LUTs		FFS	
	Min	Max	Min	Max	Min	Max
Required for Both Systems						
OPB (On-Chip Peripheral Bus)	46	436	8	668	5	145
OPB HWICAP	120	128	213	224	152	155
OPB BRAM Controller	25	34	16	30	33	55
OPB DDR SDRAM Controller	332	563	353	637	314	444
Total	523	1161	590	1559	504	799
Required for PowerPC System						
PLB (Processor Local Bus)	223	1645	270	2540	59	484
PLB to OPB Bridge	595	836	535	823	547	812
Processor System Reset Module	N/A	N/A	37	57	52	82
PowerPC (Wrapper)	0	0	0	0	0	77
Total	818	2481	842	3420	658	1455
Required for MicroBlaze System						
2 x LMB (Local Memory Bus)	N/A	N/A	0	353	0	0
2 x LMB BRAM Controller	N/A	N/A	6	6	2	2
Total	-	-	6	359	2	2
Additional Features						
OPB UART Lite	N/A	N/A	88	108	48	57
OPB Timebase WDT	N/A	N/A	63	63	111	111
Microprocessor Debug Module	67	188	45	292	79	204
Total	67	188	196	463	238	372

IPs for PowerPC system and MicroBlaze system. Non-essential IPs are provided in 'Additional Features' section.

Chapter 7: Conclusion

In this thesis we successfully realized a method for performing secure partial reconfiguration of FPGAs by implementing a special configuration controller using embedded processor cores. Under software control and within a single FPGA, the configuration controller was able to partially reconfigure an application system on the FPGA while the rest of device maintained correct operation. By performing partial bitstream encryption and authentication, this method improves the design security specifically for designs that benefit from partial reconfiguration. It also provides the flexibility of using arbitrary algorithms for authentication and encryption/decryption of partial bitstreams.

To test the configuration controller platform it was combined with an application system to create a partially reconfigurable design. The program running on this system was generating a pattern on the LEDs of the board. Using authenticated and encrypted partial bitstream, the configuration controller successfully partially reconfigured the application system after authentication and decryption of the partial bitstream. The experiment was judged to be successful when the new pattern was displayed on the LEDs of the board. Overall the contributions made by the research that led to this thesis can be summarized as under:

- This self-reconfiguring platform was realized for both PowerPC hard and MicroBlaze soft embedded processor cores for Xilinx Virtex-II Pro Platform FPGAs.

- The system-on-a-chip designs were created using various hard/soft IP cores provided by Xilinx Embedded Development Kit and resources available on Xilinx ML310 Evaluation Board.
- A program was developed to demonstrate that the FPGA can be reconfigured with an encrypted partial bitstream stored in an external memory using software cores for authentication and decryption.
- A partial bitstream has been generated using the difference-based flow targeting an active system placed in the FPGA besides the self-reconfiguring platform.
- Using the module-based flow a partially reconfigurable design intended as a proof-of-concept was created to demonstrate the advantages and problems of this flow.

The tests showed that integration of the configuration controller with minimal footprint is feasible for the designs that benefit from partial reconfiguration. It also enables embedded applications to take advantage of secure dynamic partial reconfiguration without requiring external circuitry. Using software cores for authentication and decryption also removed the need for the hardware cores for these operations at the expense of lower speed for performing the operations in software. The security analysis of this scheme also showed the improvements in the design security.

Improving the ICAP control logic from software to hardware planned by Xilinx will also enhance the performance of self-reconfiguring platforms since there will be less communication over the system bus and less processor involvement. Even though the difference-based flow involved none of the difficulties and restrictions of module-based flow it is not suitable for large designs where large blocks of logic are under

reconfiguration. To increase the ease of use for designers and decrease the development time a simple methodology along with more support and automation from tools are needed for implementation of a partially reconfigurable design using module-based flow.

To improve the performance of the current work in future extensions, synthesizable Intellectual Property (soft IP) cores which can be readily incorporated into an FPGA can be used for faster authentication and decryption. An embedded OS can also facilitate the process.

Bibliography

[1] S. Wong, S. Vassiliadis, and S. D. Cotofana, "Future Directions of (programmable and reconfigurable) Embedded Processors," in *Proceedings of the SAMOS 2002 Second International Samos Workshop on Systems, Architectures, Modeling, and Simulation*, July 2002.

[2] T. Wollinger, J. Guajardo, and C. Paar, "Security on FPGAs: State-of-the-art Implementations and Attacks," in *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 3, August 2004, pp. 534–574.

[3] Xilinx, Inc. [Online]. Available: http://www.xilinx.com/

[4] R. Krueger, "Using High Security Features in Virtex-II Series FPGAs," Xilinx, Inc., Xilinx Application Note 766, version 1.0, July 2004.

[5] B. Blodget, P. James-Roxby, E. Keller, S. McMillian, and P. Sundararajan, "A Self-reconfiguring Platform," in *International Conference on Field Programmable Logic*, Lisbon, Portugal, Sept. 2003.

[6] T. Kean, "Secure Configuration of Field Programmable Gate Arrays," in *Proceedings of 11th International Conference on Field-Programmable Logic and Applications*, Belfast, United Kingdom, 2001, FPL'01.

[7] L. Bossuet, G. Gogniat, and W. Burleson, "Dynamically Configurable Security for SRAM FPGA Bitstreams," in *Proceedings of 11th Reconfigurable Architectures Workshop*, Santa Fe, USA, 2004, RAW'04.

[8] *Virtex-II Platform FPGA Handbook*, Xilinx, Inc., 2004, version 2.0.

[9] *PowerPC Processor Reference Guide*, Xilinx, Inc., 2003, version 2.0.

[10] *Virtex-II Platform FPGA User Guide*, Xilinx, Inc., 2005, version 4.0.

[11] P. Butel, G. Habay, and A. Rachet, "Managing Partial Dynamic Reconfiguration in Virtex-II Pro FPGAs," *Xilinx Xcell Journal*, Fall 2004, www.reconf.org.

[12] "Two Flows for Partial Reconfiguration: Module Based or Difference Based," Xilinx, Inc., Xilinx Application Note 290, version 1.2, July 2004.

[13] *Development System Reference Guide*, Xilinx, Inc., 2005.

[14] *Embedded System Tools Reference Manual*, Xilinx, Inc., 2004, version 3.0.

[15] *Platform Studio User Guide*, Xilinx, Inc., 2004, version 3.0.

[16] IBM Inc. [Online]. Available: http://www.chips.ibm.com/products/coreconnect

[17] *Processor IP Reference Guide*, Xilinx, Inc., 2004.

[18] B. Gladman, "Cryptographic Implementations." [Online]. Available: http://fp.gladman.plus.com/cryptography_technology/index.htm

[19] *EDK OS and Libraries Reference Manual*, Xilinx, Inc., 2004, version 3.0.

[20] "Partial Reconfiguration on Xilinx Devices." [Online]. Available: http://www.itee.uq.edu.au/ listarch/partial-reconfig/

Appendix A: Source Code of the Configuration Controller Software

This is the main (top-level) source code for the program that is being executed on the configuration controller. The program is responsible for loading the partial bitstream from external memory, authentication, decryption, and partial reconfiguration of the FPGA.

```
extern "C" {

#include <xhwicap.h> #include <aes.h> #include <xtime_l.h>

}

#include <stdio.h> #include <string.h>

#include "aeststst.h" #include "hmac.h"

#define DEVICEID        XHI_READ_DEVICEID_FROM_ICAP

typedef unsigned char   byte; typedef unsigned long   word;
```

```
/* store the partial bitstream in a safe address in memory
 */
#define PARTIAL         0x01000000 #define ENC_PARTIAL
0x01100000 #define DEC_PARTIAL     0x01200000

/* set the number of 16-bytes blocks in the partial bitstream
 */
#define NUM_BLOCKS      882

#ifdef AES_1_BLOCK #define do_enc(a,b,c,d) f_enc_blk(a, b, c)
#define do_dec(a,b,c,d) f_dec_blk(a, b, c) #else #define
do_enc(a,b,c,d) f_ecb_enc(a, b, c, 1) #define do_dec(a,b,c,d)
f_ecb_dec(a, b, c, 1) #endif

void oblk(char m[], byte v[], word n = 16) {
    xil_printf("%s", m);

    for(word i = 0; i < n; ++i)
        xil_printf("%02X", (word)v[i]);

    //print("\r\n");
}

byte exh[32] =  // hex digits of stored key {
```

```
    0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
    0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c,
    0x76, 0x2e, 0x71, 0x60, 0xf3, 0x8b, 0x4d, 0xa5,
    0x6a, 0x78, 0x4d, 0x90, 0x45, 0x19, 0x0c, 0xfe
};

byte stored_mac[20] =  // hex digits of stored MAC {
    0x99, 0x9a, 0x59, 0xaa, 0xe8, 0x3a, 0xb5, 0xeb, 0x7b, 0xa1,
    0xa4, 0xf1, 0x58, 0x1e, 0x03, 0xe5, 0x3f, 0x68, 0xcd, 0xcc
};

int main (void) {

   XHwIcap hwicap;      /* HWICAP structure      */
   XStatus status;      /* Return value          */

   XTime tStart, tEnd;
   unsigned int tComp;

   byte    mac[20], err = 0;

   unsigned char * const ptr1 = (unsigned char *) PARTIAL;
   unsigned char * const ptr2 = (unsigned char *) ENC_PARTIAL;
   unsigned char * const ptr3 = (unsigned char *) DEC_PARTIAL;
```

```
Xuint32       * const ptr4 = (Xuint32 *) DEC_PARTIAL;

print("-- Entering main() --\r\n");

/* Set the device type
 * Before using the opb_hwicap, one must initialize it
 */
status = XHwIcap_Initialize(&hwicap, 0, DEVICEID);

if (status == XST_DEVICE_IS_STARTED) {
   print("Device is already initialized.\r\n\n");
} else if (status != XST_SUCCESS) {
   xil_printf("Failed to initialize: %d\r\n", status);
   exit(-1);
}

gen_tabs();

f_ectx  alge[1];
f_dctx  algd[1];

memset(&alge, 0, sizeof(aes_encrypt_ctx));
memset(&algd, 0, sizeof(aes_decrypt_ctx));
```

```
    /* Performing Authentication using HMAC-SHA1 algorithm
     */


}
```

Curriculum Vitae

Amir Sheikh Zeineddini was born in Tehran, Iran in 1975. He grew up and lived there till he was 26. He graduated from Azad University, Tehran in February 2000 with a B.S. in Computer Engineering. Following this, he decided to continue his studies and obtain an M.S. degree in Computer Engineering. To this end, Amir accepted admission at George Mason University in the Spring of 2003 and started working in the FPGA and ASIC Design Laboratory under the guidance of Dr. Gaj.