8-17-2013

Design Modifications and Platform Implementation Procedures for Supporting Dynamic Partial Reconfiguration of FPGA Applications

Sean Gabriel Owens

Design modifications and platform implementation procedures for supporting dynamic partial reconfiguration of FPGA applications

By

Sean Gabriel Owens

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Electrical and Computer Engineering
in the Department of Electrical and Computer Engineering

Mississippi State, Mississippi

August 2013

Design modifications and platform implementation procedures for supporting dynamic partial reconfiguration of FPGA applications

By

Sean Gabriel Owens

Approved:

Thomas H. Morris Assistant Professor Electrical and Computer Engineering (Director of Thesis)	Yoginder S. Dandass Associate Professor Computer Science and Engineering (Committee Member)
Bryan A. Jones Associate Professor Electrical and Computer Engineering (Committee Member)	James E. Fowler Professor Electrical and Computer Engineering (Graduate Program Director)
Jerome A. Gilbert Interim Dean of the Bagley College of Engineering	

Name: Sean Gabriel Owens

Date of Degree: August 17, 2013

Institution: Mississippi State University

Major Field: Electrical and Computer Engineering

Major Professor: Dr. Tommy Morris

Title of Study: Design modifications and platform implementation procedures for supporting dynamic partial reconfiguration of FPGA applications

Pages in Study: 141

Candidate for Degree of Master of Science

Dynamic partial reconfiguration of FPGAs allows systems to autonomously alter sections of their design during runtime based on the state of the system. This functionality provides size, weight, and power benefits that are useful in extreme environments such as space. Therefore, NASA has requested research into the feasibility of using a commercial off-the-shelf software flow to convert a static HDL design to support partial reconfiguration. This project presents an analysis of this conversion process using the Xilinx Partial Reconfiguration Flow to convert the static design for the ITU G.729 Voice Decoder. This paper explores the design modifications that must be made to allow for partial reconfiguration. Furthermore, an in-depth description of how to set up the hardware platform to support the HDL application is provided. Finally, timing and size data are presented and analyzed to empirically show the benefits and limitations of using dynamic partial reconfiguration.

DEDICATION

This work is dedicated to my sister, Stephanie Lynn.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. DGE-0947419 at Mississippi State University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

I would like to first thank the Mississippi Space Grant Consortium and National Science Foundation for funding the research that made this project possible. I would also like thank my family, Sherryl, John, Perryl, Jim, Evie, Jill, Justin, Dave, Matthew, and Michael for always supporting me throughout my childhood and my graduate school career. I also want thank my friends Hannah W., Hannah B., Kasey, Erin J., Erin A., and Leilani for always rooting for me. I want to especially thank Matt B. for being there for me when I needed help or to keep pushing me when I needed motivation. Finally I would like to thank the members of the CODEC team whose work was critical in being able to do this work: Zach, Troy, Mike, Parker, Nick, Cooper, Josh, David, Corey, Walter, Doug, Jeff, and Seth.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ASIC	Application-Specific Integrated Circuit
BSB	Board Support Package
CLB	Configurable Logic Block
CODEC	Coder/Decoder
COTS	Custom Over-The-Shelf
DMA	Direct Memory Access
DPR	Dynamic Partial Reconfiguration
DSP	Digital Signal Processor/Processing
EDK	Embedded Development Kit
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HWICAP	Hardware Internal Configuration Access Port
ICAP	Internal Configuration Access Port
IDE	Integrated Development Environment
IP	Intellectual Property
ISE	Integrated Software Environment
MHz	Megahertz
PAR	Place and Route

PLB	Processor Local Bus
PR	Partial Reconfiguration
RM	Reconfigurable Module
RP	Reconfigurable Partition
SDK	Software Development Kit
XPS	Xilinx Platform Studio

CHAPTER I

INTRODUCTION

1.1 Background

Scientists are constantly exploring new and more extreme environments. As the complexity of the systems required to operate in these environments increases, new methods must be researched to provide sufficient and efficient computing power. Of these environments, space has become a major area of exploration over the last half century. Space exploration introduces new challenges not found on Earth. Due to the infeasibility of having access to hard line support systems such as power, life support, and communication lines, space vehicles must operate on a finite and limited supply of power in a finite and limited space. Therefore, it is imperative that the SWAP (size, weight, and power) principle be taken into account when developing applications for execution in space.

There are three main types of integrated circuits available for application solutions. The first is the Application-Specific Integrated Circuit (ASIC) that requires custom circuit design. The level of customization of these ICs can range from fully custom to standard cell or gate array designs. However, using ASICs trades high development cost and usage flexibility for better, “application specific” performance. On the other end of the spectrum is the general purpose IC. This group of ICs includes chips such as general purpose microprocessors, as well as general purpose digital signal

processors (DSPs) and some common logic configurations. This option reduces cost and performance by allowing for the use of generalized instructions and high performance mathematics circuits. The final category of ICs is field-programmable ICs. This group provides a compromise between the two previous options. It includes Field-Programmable Gate Arrays (FPGAs) which allow controllers to define a circuit for the chip and then change the circuit and reprogram the chip with the new design. As FPGAs have grown in size, become more powerful, and become more affordable, their ability to support customized hardware designs as well as provide the ultimate flexibility of being able to be reconfigured has led the platform to become a popular development option [1].

One area in particular that has been proposed by NASA is the use of FPGAs to maintain voice communication protocols. A project was funded through an ESMD Space Grant, Senior Design Project (ID: JSC4-36-SD), to develop an HDL implementation of the ITU G.729 Voice CODEC currently in use by the space program. This project was completed by four senior design teams at Mississippi State University. Detailed lists of the contributions from each team are given in their respective design reports [8]-[11].

The G.729 is an audio compression system that produces highly-intelligible audio in low-bandwidth environments using a “conjugate-structure algebraic-code-excited linear prediction” algorithm at a rate of 8 kbits/s [23]. The CODEC operates in two stages, the encoding stage and the decoding stage, as shown in Figure 1.1. During the encoding stage, raw audio samples are streamed to the system. These samples are collected and grouped into frames that correspond to 10 ms of audio or 80 samples. The Encoder stage outputs 16-bit encoded bitstream that can then be transmitted to the destination system that houses the decoding stage. The Decoder accepts the bitstream in

groups of 80 32-bit sign-extended double-words, and outputs an 8 kHz PCM raw audio stream.

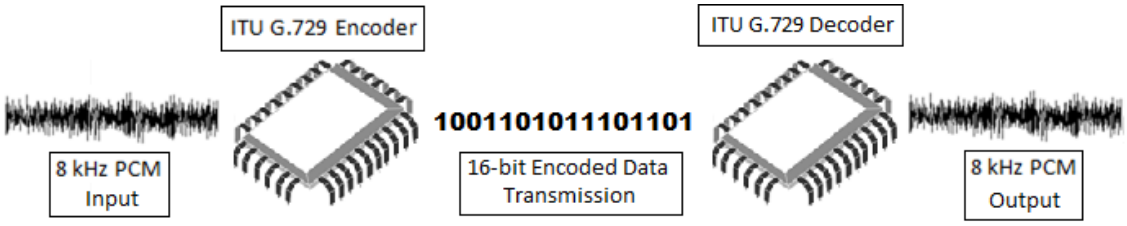

Figure 1.1 ITU G.729 Operation Overview [25]

While understanding of the mathematical and digital signal processing theory for audio coding is needed to implement the two stages as demonstrated by Owens et al. in [8]-[11], it is beyond the scope of this project. However, an overview of the architecture and functionality of the HDL implementation is necessary for understanding the case study presented in this paper. Both the Encoder and Decoder stages use the same general architecture diagrammed in Figure 1.2. The system consists of three main blocks: the control block (Top Level FSM), the datapath block (Top Level Datapath), and the interconnect between the two and the interface to the external world (Top Level Interface). The Encoder/Decoder functions by performing a set of algorithms on the input data. For the Decoder, the set consists of 36 distinct functions. These functions are coded into individual Verilog modules and called sequentially by the control FSM. All of the algorithms use a shared set of math units and memory space that are located in the Datapath block. Therefore, a multiplexer bank is used in the Datapath to direct the signals from the currently executing function to the proper math modules and memory interfaces.

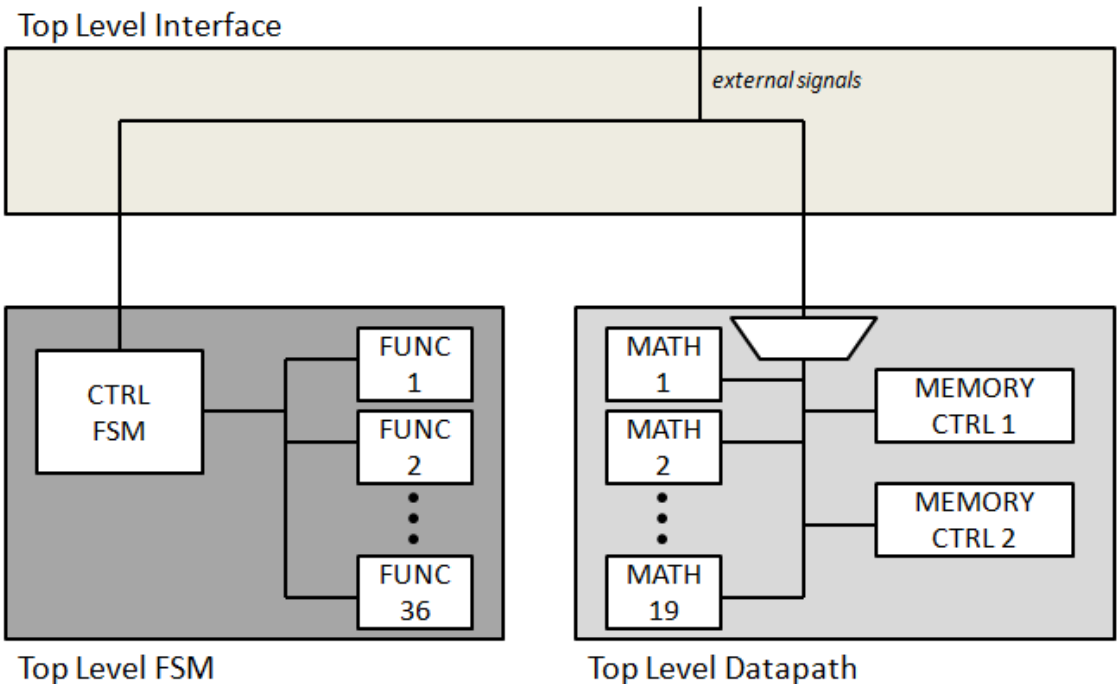

Figure 1.2 G.729 Decoder HDL Architecture

1.2 Motivation and Problem Statement

FPGAs were introduced in the late 1980's by Xilinx [4]. The chips were comprised of a set of distinct, configurable logic blocks (CLBs) that are connected by programmable switches. Initially, FPGAs could only be reconfigured on a whole chip basis. While this functionality did not lend itself to the in-application flexibility of later models, this ability still provided much more flexibility than static VLSI designs. The ability to partially reconfigure an FPGA first became available with the release of the Xilinx 6200 series [5]. Furthermore, the Xilinx Virtex-II allowed the partial reconfiguration of individual columns of the FPGA. However, as demonstrated by Sedcole et al. [6], this method of partial reconfiguration puts heavy constraints on routing. With Xilinx's release of the Virtex 4, the size of the reconfiguration frame was

reduced to a height of only 16 configurable logic blocks (CLBs) which remedied the routing problems found in the Virtex II. This ability to access small portions of the chip is achieved through the use of a “Configuration Memory Layer” as shown in Figure 1.3.

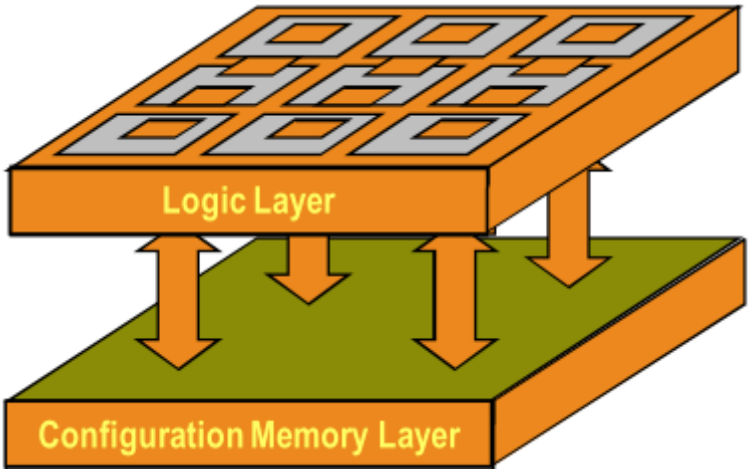

Figure 1.3 FPGA Configuration Memory

The development of techniques to allow for the partial reconfiguration of FPGAs has increased interest in using them as a platform for more diverse applications. A key feature of partial reconfiguration is that a subset of the logic programmed to an FPGA can be modified without affecting the operation of the rest of the logic on the chip [2]. This idea is illustrated in Figure 1.4 where the logic circuits defined for the *Logic Block 1* region can be replaced with the circuits defined in *Logic Block 2* without modifying or interrupting the execution of the circuits defined in the *Static Logic* region. For the rest of this report, the area of the FPGA designated to be reconfigured independently from the rest of the system will be referred to as the reconfigurable partition (RP). Furthermore, the set of circuit designs that can be interchanged with one another inside of a given

reconfigurable partition will be referred to as reconfigurable modules (RMs). These reconfigurable modules are defined by "partial bitfiles" that only give configuration information for the reconfigurable partition, as opposed to "full bitfiles" which define the configuration of the entire FPGA.

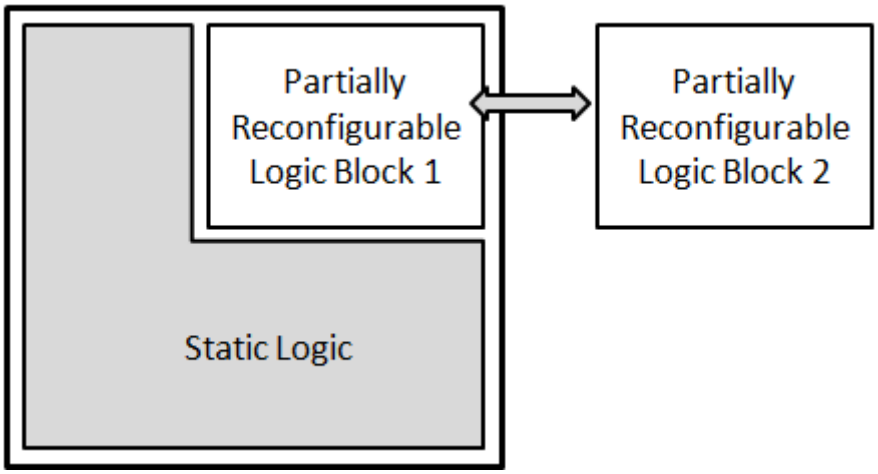

Figure 1.4 FPGA Partial Reconfiguration

The functionality of partial reconfiguration has led to the concept of Dynamic Partial Reconfiguration (DPR). Dynamic partial reconfiguration allows a system to autonomously reconfigure sections of its design based on the state of the system [3]. This allows partial reconfiguration to be utilized in real time during the operation of a system. There are two main options for configuring FPGAs: externally over a communications channel or internally using on-chip logic to modify the FPGA's logic. The external option can be useful for debugging purposes but is not sufficient to support dynamic partial reconfiguration designs. The second, internal option does support dynamic partial reconfiguration and for certain Xilinx chips utilizes a native hardware construct named

the Internal Configuration Access Port (ICAP). The ICAP interfaces through a communication FIFO that transfers reconfiguration data from the partial bitfiles in system memory to the ICAP circuit.

There are many apparent benefits to using a dynamic partial reconfiguration design. The first is design size. If a design would normally require 5 resource units, but 4 of these units are mutually exclusive, a dynamic partial reconfiguration design would only require 2 resource units on the actual chip. A second apparent benefit is power saving. This savings is due to the reduction of onboard circuitry needed to support different applications simultaneously (e.g. a separate set of chips/interfaces solely for voice communication, life support, navigation, etc.). Testing the power savings by running partial reconfiguration as opposed to the standard circuitry is beyond the scope of this project.

The benefits of dynamic partial reconfiguration have led to the conclusion that partially reconfigurable systems would be useful in environments that have space and power constraints. Therefore, dynamic partial reconfiguration has been a major area of research for space applications. Osterloh et al. [7] provide an overview of the design considerations necessary to use an FPGA in space. However, their focus is on the physical hardware considerations and data integrity aspects of the system, rather than the software capabilities of using dynamic partial reconfiguration. Recently, further research was requested by NASA to answer the question, “Can a Commercial Over-The-Shelf (COTS) design flow be used to convert a statically configured system to use dynamic partial reconfiguration?”

1.3 Previous Research and Significance of Work

Several authors have presented research that answers different parts of this question. The first, by Manet et al. [12], provides an in-depth analysis of the use of dynamic partial reconfiguration in signal processing applications. The authors provide a method for improving on the ICAP design by writing their own custom configuration management system that utilizes direct memory access (DMA) and other optimization techniques. The authors conclude that while dynamic partial reconfiguration has potential for use in signal processing, custom hardware reconfiguration controllers must be created to support the small processing period required by signal processing applications. While the authors did provide conclusions on the shortcomings of the Xilinx software flow, some of these shortcomings are outdated and the authors did not provide a detailed assessment of the software flow's capabilities for use without the need for custom interface designs.

Another study utilizing partial reconfiguration for signal processing was performed by Claus et al. [13]. The authors used partial reconfiguration to insert and remove different "hardware accelerator engines" to support the varying needs of the different functions that are running on the chip. This design is opposite of the design presented in this project where the support modules are statically configured and the function logic is reconfigured. Furthermore, the ICAP control was also modified in this study to produce better throughput for partial reconfiguration.

Another research project, by McDonald [14], attempts to use partial reconfiguration for a "software-defined radio" utilizing Xilinx's work flow. In this case, the author chose to use partial reconfigure to swap between the encoder and decoder

based on the current needs of the system. While the author does provide timing information, no indication is made as to whether timing constraints were met. The solution provided was to allocate enough memory as a buffer to avoid any loss of data due to the latency of the system.

Another investigation into the use of DPR at a higher design level was presented by Bhandari et al. [15]. The main focus of this paper was to evaluate the benefits of using the dynamic partial reconfiguration to support multiple types of signal processing systems on the same chip. The conclusions made by the authors are observations of the apparent benefits of using dynamic partial reconfiguration over static designs. However, they do provide empirical timing information for the use of dynamic partial reconfiguration in real-time signal processing, but these results are for different CODECs than the ITU G.729 CODEC used by NASA and are for changing between CODECs rather than optimizing a single one.

Based on the information provided by Xilinx and the research presented above, it was hypothesized that Xilinx's dynamic partial reconfiguration flow was sufficient to provide a COTS solution for converting statically designed systems to take advantage of the benefits dynamic partial reconfiguration offers. The ITU G.729 Decoder HDL design created by Owens et al. was chosen as a test case to evaluate the conversion process.

In order to validate this hypothesis, three stages must be completed: modify the original HDL to support dynamic reconfiguration, create hardware and software testing platforms for the application, and test the system to verify functionality, measure size, and ensure it meets application timing constraints.

For the first stage, this project provides an evaluation of the process required to convert a statically configured HDL design into a dynamically, partially reconfigurable HDL design. This includes design requirements, such as port abstraction, as well as design considerations, such as the system architecture and reconfigurable module sets.

To address the second stage, this project presents a validation platform design, as summarized in Figure 1.5. The FPGA is externally programmed with the Xilinx iMPACT tool via the JTAG cable. The JTAG is also used as the communication bus between the Microblaze soft processor and the Xilinx SDK environment where debugging information is output. The Microblaze is connected to a system bus that communicates back and forth with the test application and the HWIcap, a standard interface to the ICAP provided by Xilinx. The HWIcap is the construct that reconfigures the Decoder PR Region, which is indicated by the dotted line. The Microblaze runs a software test program that starts the Decoder, performs timing analysis, and runs the dynamic partial reconfiguration procedure when necessary. The process for creation of the platform and the design considerations associated with it are expanded upon in section 3.1.

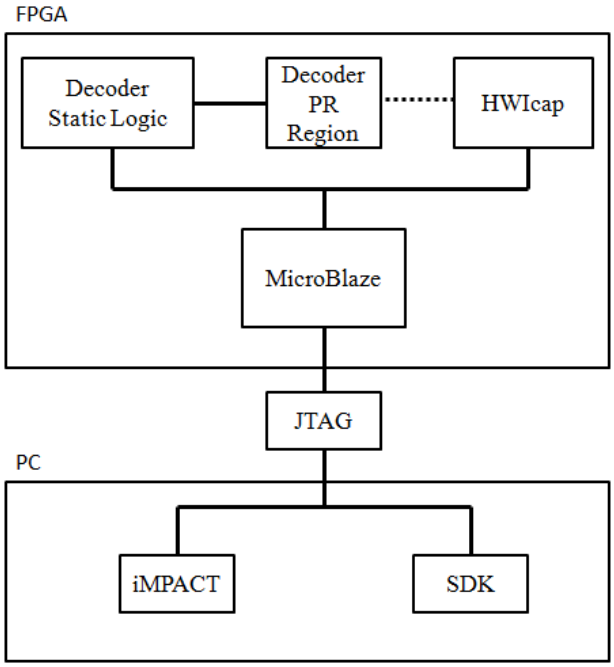

Figure 1.5 DPR Testing Platform

Finally, this project gives validation data for three aspects of the system. First, after the HDL design is modified to support dynamic partial reconfiguration, it must be tested to ensure that it still functions properly. The results of these tests and the procedures for doing so are presented in this paper. Second, although it is assumed that the size of a design will reduce when utilizing dynamic partial reconfiguration, an analysis of the actual resulting sizes is presented. This allows for an empirical conclusion as to the benefits of dynamic partial reconfiguration with respect to design size. Finally, although dynamic partial reconfiguration has many positive aspects, one property that is negatively affected is run time. Therefore, this paper presents timing data and analysis to determine if a system is capable of performing in its time constraints if it is using dynamic partial reconfiguration.

CHAPTER II

HDL DESIGN MODIFICATION

A key contribution of this project is the evaluation of the additional work in the design phase of the application development cycle necessary to produce a dynamically, partially reconfigurable system. This chapter addresses the modifications that must be made to a static design configuration as well as other aspects of the design that must be considered when developing a dynamic partial reconfiguration application. The five considerations presented below are: extracting module information for use in future design decisions, defining the set of reconfigurable modules, abstracting the port lists of reconfigurable modules, modifying the system architecture to support dynamic partial reconfiguration, and performing functional simulation to ensure the modified design produces the same results as the original.

2.1 Module Data Extraction

The first step in converting a static application to a dynamic partial reconfiguration application is to analyze the design and determine the set of functional blocks that can be implemented into independent modules. As is the case with the Decoder design, the application may already be separated into mutually exclusive modules. Once the set of modules is identified, information about each is extracted to provide an overview of the system that is necessary for consideration in future design

decisions. This process was completed by hand; however some of the information extracted could have been gathered using custom parsing programs. The key characteristics to observe are: the system's module tree, each module's instantiation level, and a breakdown of each module's port list. Another data set that is included in the module data extraction is a breakdown of the FPGA resources needed by each module. However, this information can only be obtained after the reconfigurable module synthesis process described in section 3.1 is completed.

The system module tree is a hybrid graph that shows multiple dimensions of the application flow at the same time. Vertically, it shows the sequential order in which the Decoder modules execute. Horizontally, it shows the module hierarchy. The left most modules are the highest level or top level of module hierarchy. The right most modules are the lowest level of module hierarchy, and modules listed more than once in the vertical list are functions which are used more than once to compute the overall function. This information is useful for determining the reconfigurable module set described in the next section. The next important property is each module's instantiation level. This property is derived from the system module tree. A module's instantiation level is defined as the number of modules a signal must travel through to reach the highest hierarchy or "top" level. Module instantiation levels are vital for determining the reconfigurable module set and will be examined further in section 2.2. As stated, a breakdown of each module's port list is also included in the module extraction data. This data includes counts for every port width for inputs and outputs and is necessary for the port abstraction process described in section 2.3. Table 2.1 shows the system module tree for the Top Level FSM block of the Decoder. Each column corresponds to the different

module instantiation levels. All modules of a given level are instantiated by the first module encountered above it in the next highest level.

Table 2.1 ITU G.729 Decoder System Module Tree

0	1	2	3	4	5
Top_Level_FSM					
	bits2prm_ld8k				
		bin2int			
	CheckParityPitch				
	D_lsp				
		Lsp_iqua_cs			
			Lsp_get_quant		
				Lsp_expand_1_2	
				Lsp_prev_compose	
				Lsp_prev_update	
					copy
				Lsp_stability	
			Lsp_prev_extract		
			Lsp_prev_update		
			copy		
		lsf_lsp2			
	int_qlpc				
		LSP_to_Az			
			get_lsp_pol		
	copy				
	Dec_lag3				
	Pred_lt_3				
	Random				
		L_shr			
		L_add			
		L_mult			
	de_acelp				
	Dec_gain				

Table 2.1 (continued)

		Gain_update_erasure			
		Gain_predict			
			Log2		
			mpy_32_16		
			Pow2		
		Gain_update			
			Log2		
	syn_filt				
	Weight_Az				
	calc_st_filt				
		syn_filt			
		calc_rc0_h			
	filt_mu				
	scale_st				
	pst_ltp				
		Search_Del			
		copy			
		Compute_Ltp_L			
		select_ltp			
		filt_plt			
	post_process				

The Decoder design can be broken down into four different module categories: execution, utility, math, and memory. There are 36 execution modules corresponding to the 36 functions called sequentially to decode an audio frame. There are 3 utility functions that are used by the execution modules or run independently that are instantiated in the FSM block rather than the Datapath block. There are 19 distinct math operations that are broken into modules instantiated in the Datapath block. Finally, there are four memory blocks corresponding to two memory cores with a memory controller for each. The math and memory modules are shared by all of the execution and utility functions. This breakdown is shown in Table 2.2. Furthermore, the port data for all of the

modules in the Decoder design are given in Table 2.2. The columns headings under Input and Output Port Counts correspond to the bit widths of the ports (i.e. 1, 4, 6, etc.).

Table 2.2 ITU G.729 Decoder Module Port Data

			Input Port Counts*						Output Port Counts*					
Type	Code	Module Name	1	4	6	12	16	32	1	2	12	16	32	Total Ports
System	t1	*Top_Level*	19	0	0	2	0	1	14	0	0	0	1	37
	t2	*Top_Level_FSM*												
	t3	*Top_Level_Data_Path*												
Execution	b1	*bits2prm_ld8k*	5	0	0	0	2	2	4	0	3	4	1	21
	b2	*bin2int*	5	0	0	0	4	1	3	0	1	5	0	19
	b3	*CheckParityPitch*	5	0	0	0	2	1	4	0	2	4	1	19
	b4	*D_lsp*	14	0	0	0	4	9	13	0	3	16	9	68
	b5	*Lsp_iqua_cs*	12	0	0	3	3	8	11	0	3	13	8	61
	b6	*Lsp_get_quant*	10	0	0	4	6	6	9	0	3	10	6	54
	b7	*Lsp_expand_1_2*	8	1	0	1	3	3	7	0	2	6	5	36
	b8	*Lsp_prev_compose*	6	0	0	5	1	4	5	0	3	6	2	32
	b9	*Lsp_prev_update*	6	0	0	2	2	2	5	0	2	4	3	26
	b10	*Lsp_stability*	7	0	0	1	2	3	6	0	2	4	5	30
	b11	*Lsp_prev_extract*	7	0	0	5	1	5	6	0	3	7	3	37
	b12	*lsf_lsp2*	9	0	0	2	4	4	8	0	3	11	2	43
	b13	*int_qlpc*	29	0	0	0	11	13	17	0	3	17	13	103
	b14	*LSP_to_Az*	27	0	0	2	10	12	16	0	2	15	13	97
	b15	*get_lsp_pol*	29	0	0	1	10	12	16	0	2	15	13	98
	b16	*Dec_lag3*	6	0	0	0	5	1	5	0	2	6	1	26
	b17	*Pred_lt_3*	8	0	0	1	4	5	7	0	3	6	5	39
	b18	*Random*	3	0	0	0	0	0	1	0	0	1	0	5
	b19	*de_acelp*	6	0	0	0	3	1	5	0	2	6	1	24
	b20	*Dec_gain*	23	0	0	0	5	9	14	0	3	16	9	79
	b21	*Gain_update_erasure*	7	0	0	0	2	3	6	0	2	5	4	29
	b22	*Gain_predict*	20	0	0	0	4	7	11	0	3	16	6	67
	b23	*Gain_update*	16	0	0	0	3	7	9	0	3	10	6	54
	b24	*syn_filt*	16	0	1	4	2	5	9	0	2	9	5	53
	b25	*Weight_Az*	6	0	0	3	1	3	5	0	2	4	3	27
	b26	*Residu*	9	0	0	3	2	5	8	0	2	9	5	43
	b27	*calc_st_filt*	25	0	0	6	8	11	17	0	2	18	11	98
	b28	*calc_rc0_h*	14	0	0	2	6	8	11	0	2	9	8	60
	b29	*filt_mu*	17	0	0	3	7	10	15	0	2	16	9	79
	b30	*scale_st*	16	0	0	3	8	8	14	0	2	13	8	72
	b31	*pst_ltp*	27	0	0	2	17	15	20	0	3	23	13	120
	b32	*Search_Del*	23	0	0	1	14	11	18	0	3	28	10	108
	b33	*Compute_Ltp_L*	10	0	0	2	6	5	9	0	3	13	6	54
	b34	*select_ltp*	16	0	0	0	18	7	11	1	0	16	6	75
	b35	*filt_plt*	8	0	0	0	6	4	7	0	2	8	4	39
	b36	*post_process*	15	0	0	10	8	10	11	0	3	13	8	78

Table 2.2 (continued)

Utility	u1	*copy*	5	0	0	2	2	2	4	0	2	2	3	22
	u2	*Log2*	8	0	0	0	2	5	6	0	1	8	4	34
	u3	*Pow2*	13	0	0	0	4	4	6	0	1	9	3	40
Math	m1	*add*	3	0	0	0	2	0	2	0	0	1	0	8
	m2	*L_add*	3	0	0	0	0	2	2	0	0	0	1	8
	m3	*sub*	3	0	0	0	2	0	2	0	0	1	0	8
	m4	*L_sub*	3	0	0	0	0	2	2	0	0	0	1	8
	m5	*mult*	4	0	0	0	2	0	2	0	0	1	0	9
	m6	*L_mult*	3	0	0	0	2	0	2	0	0	0	1	8
	m7	*shl*	3	0	0	0	2	0	2	0	0	1	0	8
	m8	*L_shl*	3	0	0	0	1	1	2	0	0	0	1	8
	m9	*shr*	3	0	0	0	2	0	2	0	0	1	0	8
	m10	*L_shr*	3	0	0	0	1	1	2	0	0	0	1	8
	m11	*norm_l*	3	0	0	0	0	1	1	0	0	1	0	6
	m12	*norm_s*	3	0	0	0	1	0	1	0	0	1	0	6
	m13	*L_abs*	3	0	0	0	0	1	1	0	0	0	1	6
	m14	*L_negate*	3	0	0	0	0	1	1	0	0	0	1	6
	m15	*L_mac*	3	0	0	0	2	1	2	0	0	0	1	9
	m16	*L_msu*	3	0	0	0	2	1	2	0	0	0	1	9
	m17	*mpy_32_16*	9	0	0	0	2	3	4	0	0	6	2	26
	m18	*Mpy_32*	9	0	0	0	1	4	4	0	0	6	2	26
	m19	*div_s*	5	0	0	0	3	1	4	0	0	3	2	18
Memory	mem1	*Scratch_Memory_Controller*	2	0	0	2	0	1	0	0	0	0	1	6
	mem2	*scratch_memory_V1*	2	0	0	2	0	1	0	0	0	0	1	6
	mem3	*Constant_Memory_Controller*	2	0	0	1	0	1	0	0	0	0	1	5
	mem4	*CONSTANT_MEM*	2	0	0	1	0	1	0	0	0	0	1	5

* The columns under the port count headings refer to the bitwidths of the different sized ports (i.e. 1-bit, 4-bit, 6-bit, etc.)

2.2 Reconfigurable Module Set Selection

The first dynamic partial reconfiguration design decision is choosing the set of modules to be included in the reconfigurable module set. The number of reconfigurable modules sets possible is bounded by $n!$, where n is the number of identified potential reconfigurable modules. This number increases very rapidly as the number of reconfigurable modules increases. Therefore, this section provides common schemes and practical constraints that will limit the number of set compositions.

To determine which modules are eligible for partial reconfiguration, all modules must be analyzed to decide if they execute sequentially or concurrently with other modules. Because sequentially executed modules only need to be present on the chip during their execution time, they are perfect candidates for partial reconfiguration. However, modules that run concurrently with other modules must be on the chip whenever the other modules are executed. Such is the case for the math and memory modules in the Decoder. While the modules are not used by every execution module, their use is often enough and their design size is small enough that the added complexity of including them in the reconfigurable module set would not be worth the benefits gained. This reduces the potential reconfigurable modules to the 36 execution and 3 utility modules.

The execution modules from Table 2.2 are each finite state machines which were derived by converting a C programming language functions to Verilog HDL. The execution modules perform complex mathematical operations using the shared math modules and read and store data from the share memory objects using the shared utility functions. The execution module hierarchy mirrors the original C program and represents the hierarchy found there. Instantiation of one module within another generally represents either a large function which was carved out as a separate subroutine or represents a function which was called multiple times in a loop. For the loop case the Verilog HDL version includes a finite state machine in the higher hierarchy level module which implements the loop and uses module to module hand shaking to wait for execution of the lower module's finite state machine. This organization had a large

impact on the decision process when choosing which modules to dynamically reconfigure.

An obvious selection scheme would be to select every execution module for the reconfigurable set. However, many execution modules utilize multiple instantiation levels. All of the modules in a "multi-instantiation level" configuration can only be implemented in three ways. The first implementation would have all of the execution modules individually use the same reconfigurable partition regardless of the module's instantiation level in Table 2.1. Often leaf modules communicate through the higher level modules to the shared math modules. As such the leaf module requires the multiplexors or wires in the instantiating modules to be present. This creates a concurrency requirement between the leaf module and its instantiating module. This makes it impossible to treat each execution module as a separate member reconfigurable module set. In effect, this creates groupings of modules which must always be simultaneously present and reduces the size of the potential reconfigurable module set accordingly. A second option considered was to move the lower hierarchy modules' logic directly into the higher level modules. This option is possible but proved impractical due to the complexity of the logic that must be integrated into the higher level state machines of the altered execution modules. The final option is to define a separate reconfigurable partition, and thereby separate reconfigurable module sets, for each instantiation level. This is infeasible because it requires the instantiation of a reconfigurable partition inside of another reconfigurable partition, which is not supported by Xilinx's PlanAhead software. Therefore, it can be concluded that partially reconfiguring every module in a

"multi-instantiation level" design is impractical when using the standard Xilinx partial reconfiguration flow.

So, to avoid a reconfigurable module set that spans across instantiation levels, the reconfigurable module set was limited to modules from a single level of hierarchy. This scenario forces all of the sub-modules associated with a given reconfigurable module to dynamically added or removed from the FPGA simultaneously with the instantiating module. Consequently, the set of possible reconfigurable modules without modifying the original HDL structure is reduced to the set of modules in the first instantiation level. For the Decoder, this set consists of 18 modules.

One of the goals of this project is to measure the impact of dynamic partial reconfiguration on design size and runtime. For design size, data for the 18 reconfigurable module set chosen above can be obtained through the unit synthesis procedure described in section 3.1. However, it is unnecessary to use the entire set to obtain runtime data. This is because the reconfiguration time is dependent on the size of the reconfigurable partition rather than the size of the reconfigurable module set. Therefore, it is sufficient to select a small subset of the reconfigurable module set and extrapolate the runtime for the whole set based on the data observed from using the subset. Moreover, because the runtime is solely dependent on reconfigurable partition size, the results can be extrapolated to predict the impact of implementing any selection scheme. The subset chosen for the Decoder application consists of the first two modules in the first instantiation level: *b1* (*bits2prm*) and *b3* (*CheckParityPitch*) as shown in Table 2.1.

2.3 Reconfigurable Module Port Set Selection and Abstraction

One design modification that is required by Xilinx for dynamic partial reconfiguration systems is the port set selection and abstraction of all modules in the reconfigurable module set. That is, all reconfigurable modules associated with a single reconfigurable partition must have identical port lists [17]. As is the case with the Decoder, most HDL systems are designed in such a way that each modules port list is unique to the needs of that module. Therefore, it is necessary to modify the port lists of all of the reconfigurable modules. There are three steps to this process: generalizing the port names, selecting the port set, and abstracting the port list in the HDL design. Three methods of port naming and set selection are presented below.

The first method is to create a "superset" of all of the ports for every reconfigurable module. This method has the benefit of being easy to implement but also has the largest port list. This can lead to the reconfigurable partition being "I/O Limited", where the size of the partition is bounded by the number of ports rather than the size of the logic contained. The first step, generalizing port names, for the superset method can be easily accomplished by adding a unique module identifier either as a prefix or suffix to all of the ports. This process is illustrated in Figure 2.1.

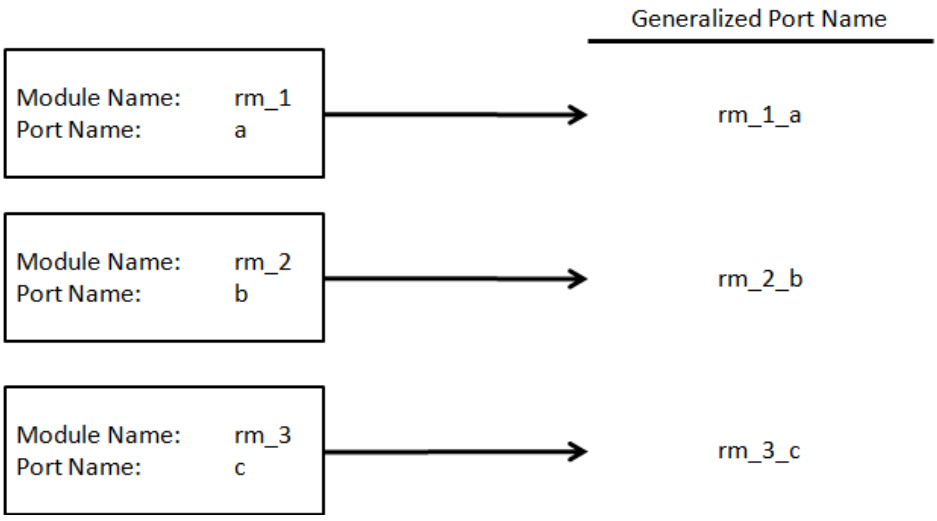

Figure 2.1 Port Abstraction – Superset Port Name Generalization

Once all of the names have been generalized, the port set can be selected. In this case, every port is selected for the superset as shown in Figure 2.2. In the figure, all three modules have the same three ports. However, using the superset method, every port is prefixed and added to the superset port list resulting in a nine port set.

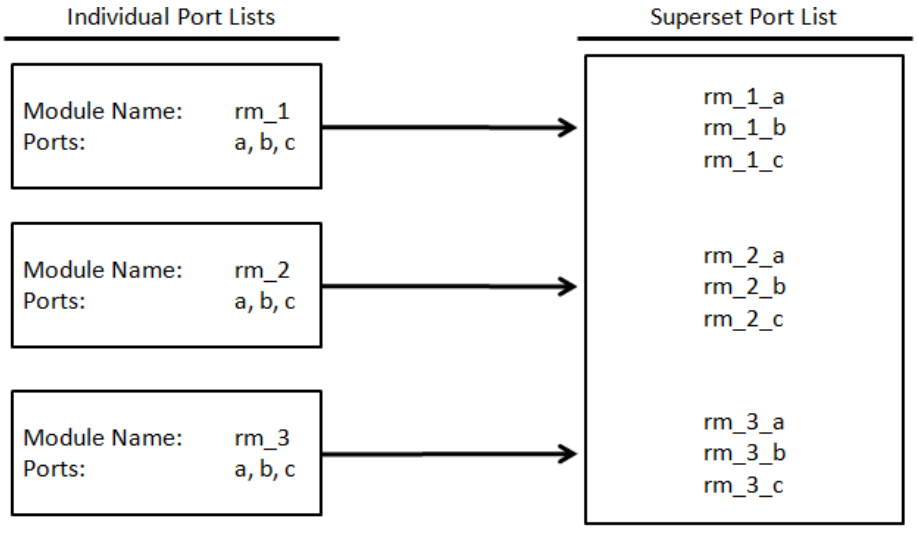

Figure 2.2 Port Abstraction – Superset Port Set Selection

This unnecessary redundancy is addressed by the second approach to port set selection. The goal of the second approach is to minimize the number of ports by defining two vectors, one input and one output, that are sized to match the largest port list in the reconfigurable module set. Then, each module's ports can be mapped into the vectors. This approach has the benefit of removing the need for port name generalization and guaranteeing the minimum port list size. However, it does require more complex logic to implement than the "superset" approach. An illustration of the "vector" approach is shown in Figure 2.3. In this example, there are three modules with varying quantities of 2-bit, 16-bit, and 32-bit ports. The calculations on the right show how large a vector would need to be to support that module. Because *rm_3* requires the largest vector, it is chosen as the vector size for the reconfigurable module port list. Although not drawn to scale, the boxes at the bottom give a representation of how each module's port list could be mapped into the vector.

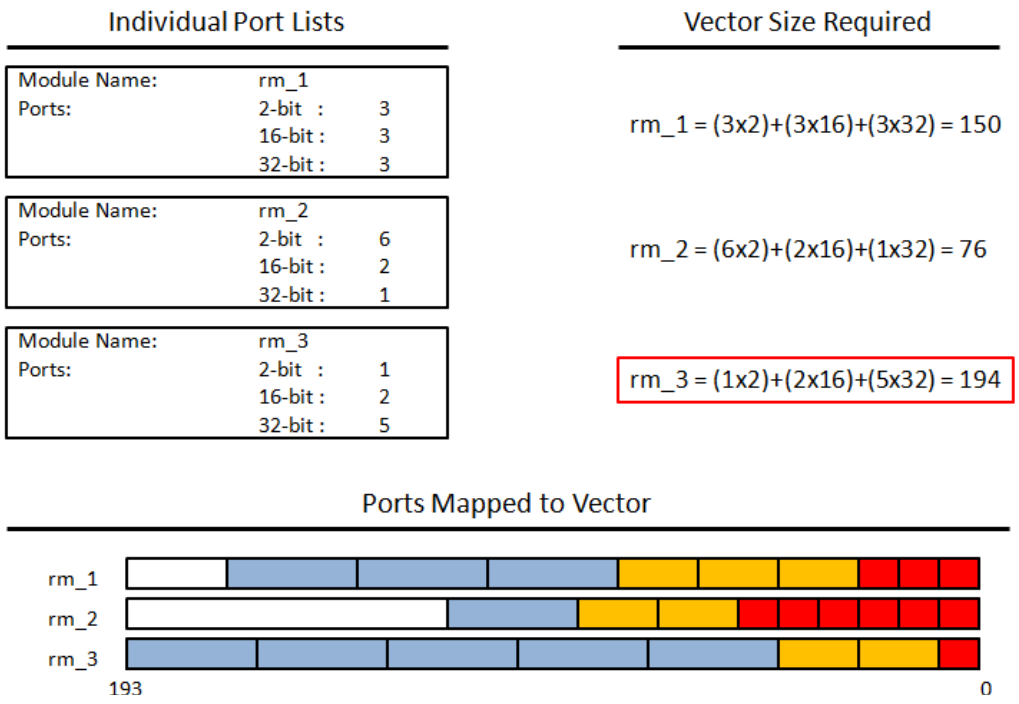

Figure 2.3 Port Abstraction – Vector Port Set Selection

A third approach, that bridges the gap between the two previous options, is to define a port set that is a common subset to all of the reconfigurable modules. To implement this "subset" approach, a different naming scheme must be used than the one used in the "superset" method. In this naming scheme, all ports are converted to a common naming convention that includes three properties: whether the port is an input or an output, the width of the port, and a unique number for that port given its type and size. For example, the third 8-bit input would have a generalized port name of the following form: *input_8_3*.

Using the module port data collected in Table 2.2, a minimum set can be created by taking the largest count for each port width and type. This procedure is illustrated in Figure 2.4. Using the same example modules from the "vector" approach, the arrows in

this figure indicate which module is responsible for contributing the most ports for each width. For the 2-bit case, module *rm_2* is the contributing module because it requires 6, 2-bit ports, whereas *rm_1* and *rm_3* only require 3 and 1, respectively. The boxes at the bottom of the figure show the calculations for each of the port widths.

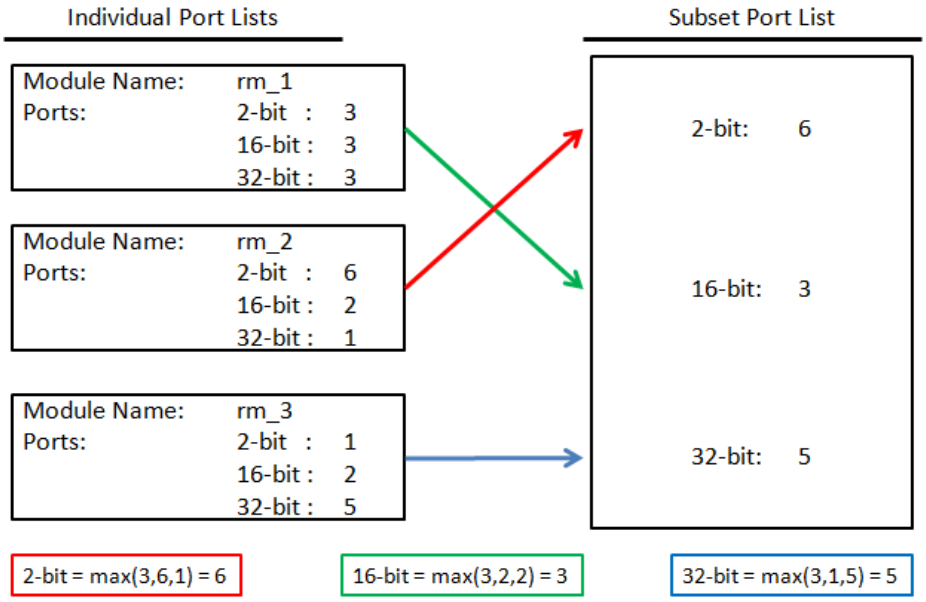

Figure 2.4 Port Abstraction – Subset Port Set Selection

This approach requires more effort to implement than both of the other methods but will result in a smaller port list than the "superset" method and a more legible design than the "vector" approach.

All of the selection schemes presented above modify the port lists for the reconfigurable modules; therefore, both require a port abstraction approach that will link the original unique port names to the new generalized port names without requiring the modification of any execution logic in the HDL design. It does, however, require

architecture modifications in both the instantiating module and the reconfigurable modules. In the reconfigurable modules, all abstracted ports can be tied to their respective signals with an assign statement. In the instantiating module, however, the abstraction process is more complicated. Any mutually exclusive output signals associated with a reconfigurable module can be linked to their respective abstracted port with an assign statement. For the rest of the signals, a multiplexer circuit is implemented to allow the system to determine what signal should be connected to what abstracted port based on which reconfigurable module is currently on the chip.

Because one of the main goals of this project is to evaluate the design size benefits associated with dynamic partial reconfiguration systems, "I/O Limiting" is a factor when choosing which option to implement. The "vector approach produces the greatest size reduction benefit possible. That is, using the approach, if the reconfigurable partition size can be reduced to the point that it is I/O limited, any further reduction in size would have to be the result of a reduction in the port list size. However, the port list size is already at its minimum, therefore reducing it would cause some reconfigurable module to not be supplied its required port counts. This creates a contradiction. The selection approach chosen for the Decoder conversion is the subset solution because of its reduced port list size and legibility. Based on the data in Table 2.2, Table 2.3 shows a comparison of the port lists created for the Decoder using all three methods. It is clear from this table that the vector method provides the best results while only sacrificing code legibility.

Table 2.3 Port Abstraction Method Size Comparison

	Superset	Vector	Subset
Number of Ports Across Boundary	976	2	139
Number of Bits Across Boundary	12,614	1643	1849

2.4 System Architecture Modification

Although it is possible for a given application to be architected in a way that already supports dynamic partial reconfiguration, this section outlines modifications that were made to the Decoder application to illustrate the type of architecture necessary for dynamic partial reconfiguration.

2.4.1 Data Flow Architecture

The first modification to the architecture is the removal of data flow selection logic that is rendered unnecessary during the partial reconfiguration conversion process. In the static design configuration of an application, a multiplexer bank is used to select the proper set of signals to route from the execution modules to the shared modules based on which module is currently executing. This original organization is shown in Figure 2.5. As can be seen in the figure, every executable module sends every one of its output to the datapath block. Because the execution modules operate sequentially, only one set of the outputs will be active at a time resulting in a high percentage of the signals being inactive at any given time. Furthermore, because the execution modules are all attempting to access the same resources in the datapath block, many of the signals redundantly point to the same destination.

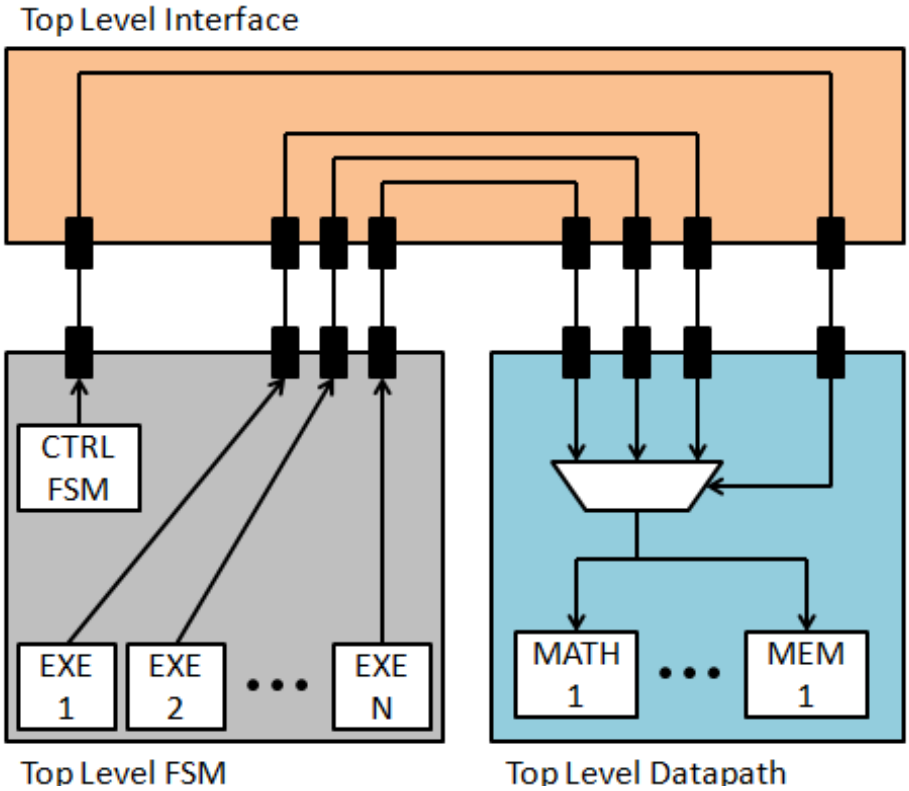

Figure 2.5 Original Data Flow Architecture

Although this design requires more inter-module communication than necessary, it functions properly. However, this design is not sufficient for a dynamic partial reconfiguration system. It is possible that all of the execution modules are designated as reconfigurable modules. In this case, only one of the executable module blocks will ever be on the chip at a time, producing only a single set of outputs. The adjustment for linking this set of outputs to the proper design signals is already managed by the port abstraction multiplexer described in the previous section. As a result, only a single set of signals connects the FSM block to the Datapath block. This makes the original shared module multiplexer useless because only a single input set will ever be driven resulting multiple floating inputs, as shown as red ports in Figure 2.6. Therefore, the multiplexer

bank in the datapath can be removed from the system architecture to reduce design complexity and size. The final system data flow architecture is shown in Figure 2.7.

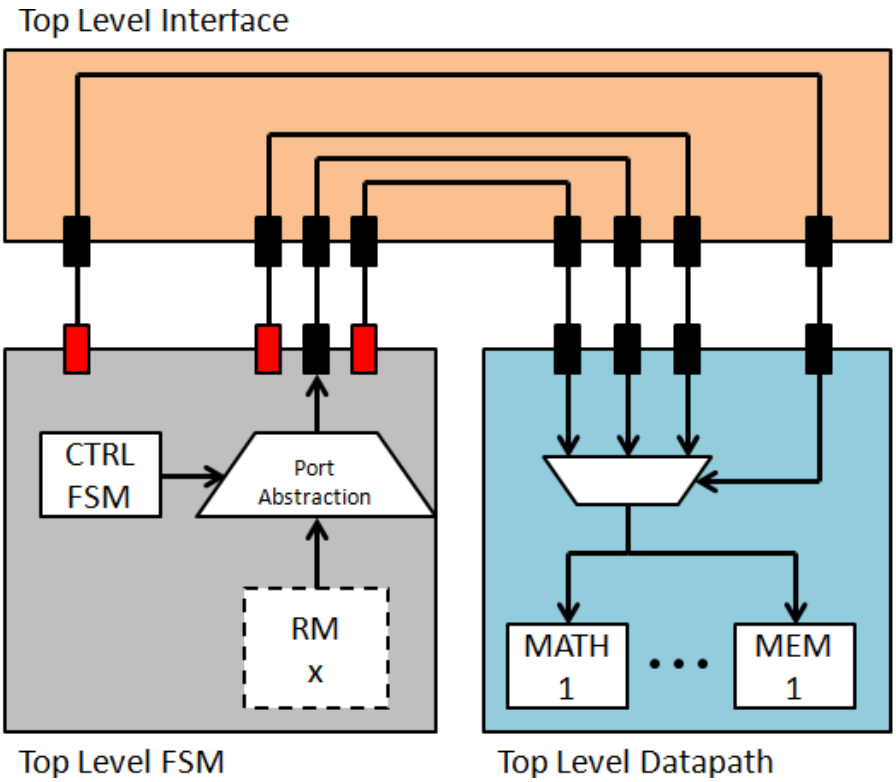

Figure 2.6 Decoder Intermediate Data Flow Architecture

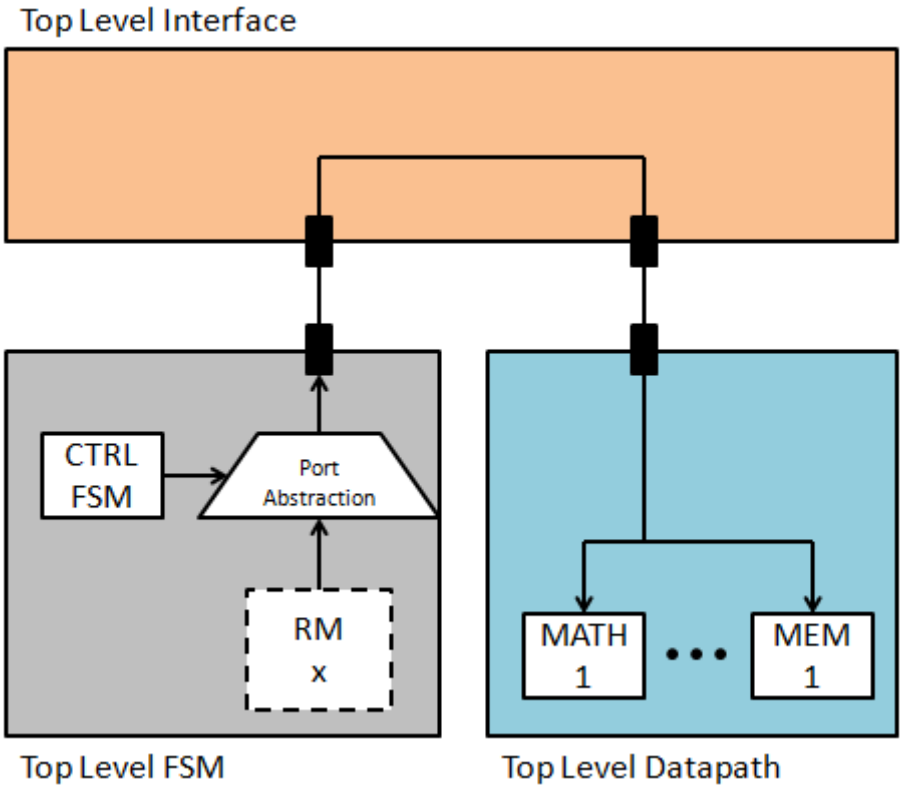

Figure 2.7 Decoder Final Data Flow Architecture

2.4.2 External Signals

The second architecture modification required to convert the Decoder application was to revise the list of external signals from the Decoder and update their functionality. The modifications in this section are the only changes that affect the operation of the Decoder from its original version. However, the validation procedures presented in section 2.5 show a method for checking that the application will still function properly. There are three alterations made to the external signals of the Decoder: the removal of testbench signals, the extension of the completion state, and the addition of partial reconfiguration signals.

The Decoder implementation provided by Duplantis et al. had many external signals that were used for application testbenching. These signals were used to pause execution at certain key points during the decoding process to allow the testbench to validate internal memory state. There signals are unnecessary now that the functionality of the Decoder's logic has been verified, so they have been removed to reduce the Decoder's design size. However, the pauses were created by states in the control logic that wait for an external signal to continue execution. To avoid modifying the control logic, a single continue signal is added to the external port list and permanently driven high. This allows the Decoder to "fall through" the pause states in the control logic.

Another modification to the external signals that, while not required to support dynamic partial reconfiguration, is necessary to interface with the Microblaze soft microprocessor is the extension of the completion state. In the original configuration, after the system completes the decoding process on one frame, a "done" state is entered for a single clock cycle that sets an external "done" signal high. After this clock cycle the system returns to the initial state and the "done" signal returns low. This single clock cycle is virtually impossible to detect by a polling trigger in the Microblaze's program, which is needed to serve the Decoder with the next set of frame data. Moreover, the timing operations performed in CHAPTER IV could only be completed if the Microblaze can detect when the Decoder has finished its operation on a frame. Therefore, the "done" signal was modified to remain at a high state while the Decoder is in its initial state and only be driven low when the start signal for the next frame is received.

The final modification to the external port list was the addition of two signals necessary to implement dynamic partial reconfiguration. These two signals are a load and

ready signal that allow the Decoder and Microblaze to communicate information about which reconfigurable module needs to be loaded and which is currently on the chip, respectively. Each signal is as wide as necessary to encode the number of reconfigurable modules in the reconfigurable module set. In the case of the Decoder, there are 24 reconfigurable modules; therefore, the load and ready signals are both five bits wide.

A key design decision for implementing dynamic partial reconfiguration is determining where in the control logic to signal the Microblaze to begin reconfiguring a new reconfigurable module. The simplest method would be to signal the Microblaze immediately before executing a reconfigurable module and then wait for the ready signal to report that the module has been loaded onto the FPGA to proceed with the execution of that module. However, simple analysis of the application flow can produce better results if the design has extra control logic in between the calls to the reconfigurable modules, as is the case with the Decoder. A system that calls functions sequentially gives the ability to determine which module will be needed next prior to execution. Therefore, it is possible to determine the earliest point in the application, after the previous reconfigurable module finishes executing, that a reconfigurable module can be guaranteed to be the next module to be executed. The control logic can signal the partial reconfiguration to begin at this point and continue executing while the partial reconfiguration process takes place in parallel. This method also requires that the application wait for the ready signal before each reconfigurable module is executed. This is necessary to guarantee that the partial reconfiguration process has completed before the application attempts to use the reconfigured logic. Figure 2.8 and Figure 2.9 show the theoretical difference in the

amount of time spent waiting on the partial reconfiguration process to complete. Note: these drawings are not to scale.

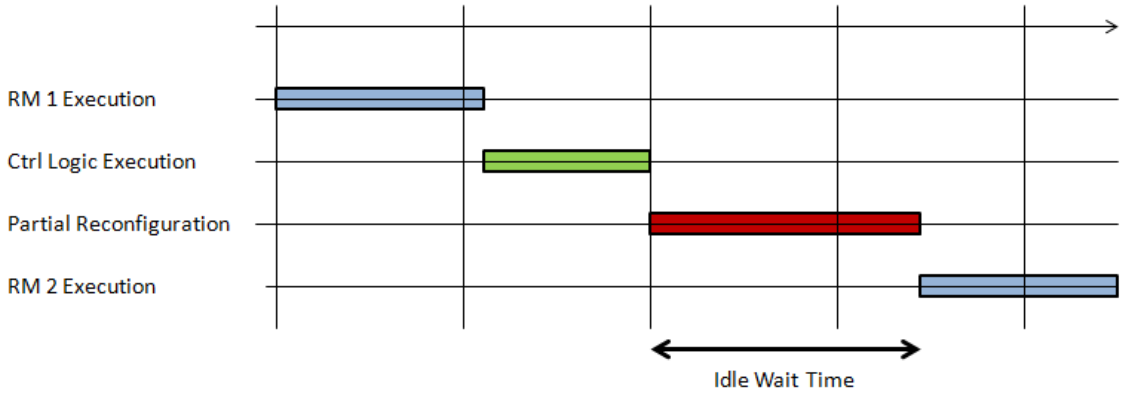

Figure 2.8 Partial Reconfiguration Wait Time Without Optimization

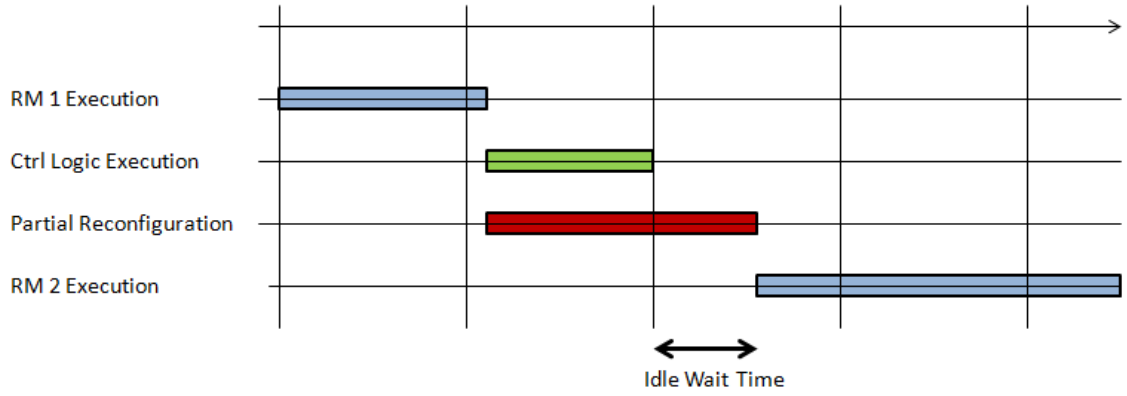

Figure 2.9 Partial Reconfiguration Wait Time With Optimization

APPENDIX A gives a state diagram of the control logic for the Decoder and an illustration of which states (red) are executing reconfigurable modules and which states (yellow) are the optimized points where the Decoder signals the Microblaze to begin the partial reconfiguration process.

2.5 System Functionality Simulation

A vital component of the HDL application development cycle is operation validation. Although an application being converted to be partially, dynamically reconfigurable is assumed to have been functionally validated, the significant modifications to the system detailed in the previous sections require that the application be revalidated to ensure that the new design produces the same output as the original. However, standard verification methods for HDL designs cannot be used when testing a dynamic partial reconfiguration system. This is due to the fact that Xilinx tools are incapable of simulating partial reconfiguration [17]. Consequently, a customized test bench is presented in this section that allows for the testing of the functionality of the partial reconfiguration design. It should be noted that, while the HDL code tested by the presented method is not exactly the same as the code that is ultimately synthesized and implemented, the differences between the two do not affect the output of the system.

The method used in this project instantiates all of the reconfigurable modules in the system rather than replacing them with a black box. A difficulty faced is rectifying the abstracted ports that are now repeated for each reconfigurable module. The redundant input signals do not cause a problem because they are all loaded from a single, shared source in the datapath. Therefore, the redundant signals act as a fan out of the single signal. On the contrary, the redundant output signals create a single net that is driven from multiple different sources, one from each reconfigurable module. This multiple sources condition creates an undefined state for the net. Different options were proposed for overcoming this dilemma that included using FORCE/RELEASE blocks to simulate the effect that only a single reconfigurable module would be operable at a time. Another

option proposed was to use the *wand* and *wor* Verilog constructs to define the rules for how the nets are driven. There is no signal conflict in the FPGA realization of the design because there is only one black box module present in the reconfigurable partition at a time. This problem was solved by assigning the output signals from each reconfigurable module to separate nets uniquely associated with each reconfigurable module. This allows modifications related to preparing the design for reconfigurable computing to be tested. The output modifications were checked by visual inspection. The results of the system functional simulation are shown in Figure 2.10. As can be seen in the figure, the Decoder begins execution at 0 milliseconds shown by the vertical yellow marker. The white marker indicates the point at which the done signal goes high. Thus, it is shown that the time taken to decode a single frame in simulation is approximately 2.4 milliseconds. The test bench periodically performs internal memory state checks to verify that the Decoder is executing properly. As a result, the same checks were performed on the new system design which completed without any memory check errors, indicating that the new system design functionally operated in the same manner as the original.

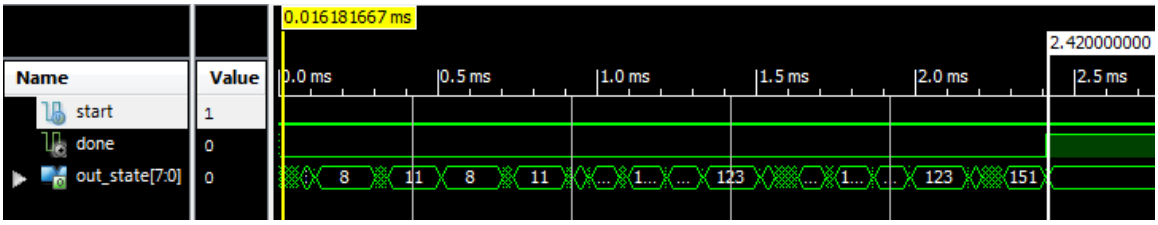

Figure 2.10 Decoder Functional Simulation

CHAPTER III

DYNAMIC PARTIAL RECONFIGURATION SYSTEM DESIGN AND IMPLEMENTATION

This chapter describes the procedures necessary to implement a dynamic partial reconfiguration system. This includes bottom-up synthesis of the reconfigurable modules, design and creation of the hardware platform, and the design of a software application for running the decoder and performing runtime calculations is presented.

3.1 Reconfigurable Module Unit Synthesis

After an application has been modified to support dynamic partial reconfiguration, the first step in implementing the system is to perform a bottom-up synthesis of the reconfigurable modules in the reconfigurable module set. Because the instantiation of these modules is replaced by a black box in the full application, the modules must be individually synthesized in order to generate netlist files that will be used by the PlanAhead tool to implement the reconfigurable design. In order to complete this process, a separate Xilinx ISE project must be created for each reconfigurable module. The option to automatically add I/O Buffers must be turned off in the Xilinx Specific Options tab of the Process Properties panel. This option is turned off because otherwise the software would attempt to map the ports of the module to the I/O pins on the FPGA. Once this is done, the modules can be synthesized and their netlist files stored

for later use. In addition, the synthesis report will provide the FPGA resource requirements necessary for reconfigurable partition size calculations.

3.2 Dynamic Partial Reconfiguration Hardware Platform

The next step in the dynamic partial reconfiguration system implementation process is to design and create a hardware platform to support the HDL application. An overview of the hardware platform designed for this project is given in section 1.3. This section will explain the process for creating this platform including discussion of the different design decisions made during the process. A tutorial version of this process is given in APPENDIX B. There are two main procedures for building the hardware platform. The first is platform design and synthesis, and the second is system configuration and implementation.

3.2.1 Platform Design and Synthesis

In order to build a hardware platform, several decisions, such as which components to include and how to connect them, must be made. All of the procedures in this section are managed through Xilinx's ISE Project Manager software. Because the design of this platform revolves around a central soft processor, the Microblaze, a template embedded processor project can be created in the Project Manager, and from the manager, the Xilinx Environment Development Kit (EDK) can be launched to customize the embedded processor. The Microblaze was chosen for this project because it is the example processor used in all of the Xilinx Partial Reconfiguration tutorials and documentation.

The Embedded Development Kit is included in the Xilinx Platform Studio. It is used to customize an embedded processor core by adding and modifying different peripherals such as memory, communication controllers, and user defined logic. As described in previous sections, the Microblaze is used in this project to serve frame data to the Decoder, launch and manage the partial reconfigurations, time the execution of the Decoder, and communicate this data back to the user. Therefore, several peripherals are needed to support this functionality. They are shown in Figure 3.1. In order to support timing operations, an external system timer was added. A custom logic peripheral was added to connect the Decoder to the system bus, and an ICAP controller was added to allow the Microblaze to interface with the configuration port. The ICAP controller, named *HWICAP*, is designed to run at maximum frequency of 100 MHz; therefore, the system clock for the entire platform was set to this frequency.

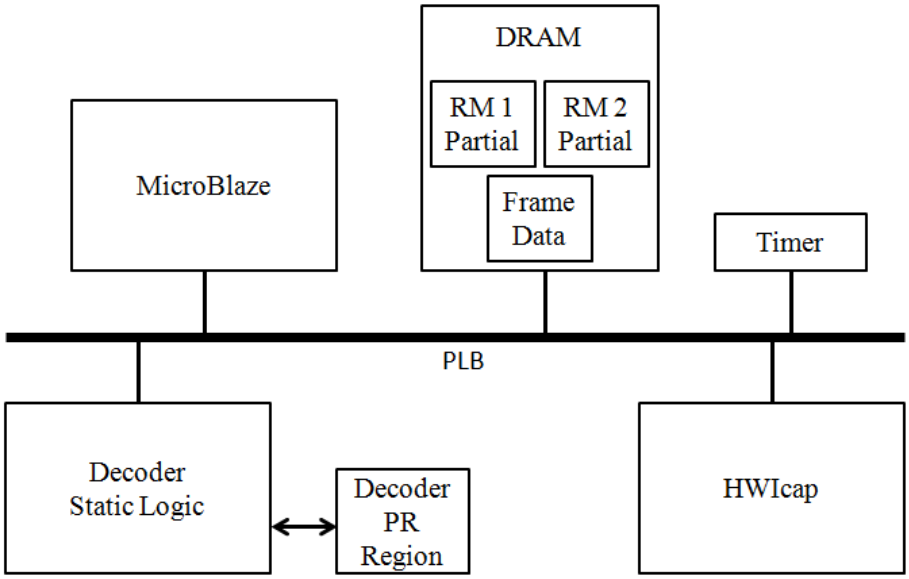

Figure 3.1 DPR System Block Diagram

Creating a custom logic peripheral requires extra steps to include in the system design. The first step is to configure the custom peripheral's bus interface and template structure. This is done inside the Xilinx EDK. After creating the peripheral, it can be opened as an independent project in a separate Xilinx ISE window. In this new instance, code is added to the template interface to instantiate the partial reconfiguration application, the Decoder for this project. Furthermore, the external signals from the application are linked to the software registers defined by the peripheral, granting the Microblaze access to these signals. Once the peripheral design is completed, it can be synthesized in the Xilinx ISE window. In order to add the new version of the custom peripheral, it must be re-imported into the Xilinx EDK project repository. From there, it can be added to the platform design in the same manner as any other stock peripheral.

After adding all of the necessary peripherals to the platform design, the platform must be synthesized to generate a system netlist file that will be used by the PlanAhead tool to configure and implement the partial reconfiguration aspects of the system. Synthesis of the platform is performed in the original Xilinx ISE instance.

3.2.2 Partial Reconfiguration Design and System Implementation

After the hardware platform is designed and synthesized, it can be configured for partial reconfiguration. This process is completed in the Xilinx PlanAhead software. Note: In order to develop and implement partial reconfiguration designs, the *Partial Reconfiguration License* must be acquired from Xilinx. For this project, the license was acquired on a trial basis for Academic research.

The first step in this process is creating a new process and selecting the system netlist file created in the previous section as the *Top Netlist File*. This will allow the

software to load the system's data for floorplanning and implementation. The next step is to define and configure the reconfiguration partition. The software will automatically detect the black box module that is the placeholder for the reconfigurable modules, so a new partial reconfiguration partition is connected to this black box entry. This allows the reconfigurable module netlist files created in the unit synthesis process to be assigned to the reconfigurable partition.

After defining the reconfigurable partition and adding the reconfigurable modules to it, a physical region on the FPGA must be defined for the partition. This is necessary to ensure that all of the reconfigurable logic is confined to a constant set of FPGA configurable logic blocks (CLBs) so that the partial reconfiguration will not interfere with the static logic located on the rest of the chip. Before the partition can be defined, its size must be calculated based on the requirements of the reconfigurable modules.

Figure 3.2 shows the resource structure for a Virtex 5 FPGA. The light pink and light blue vertical bars in the FPGA diagram correspond to BRAM and DSP resources, respectively. The dark blue regions of the FPGA correspond to CLBs which make up the majority of the chip. These CLBs are made up of two SLICE resources, either one SLICEL and one SLICEM or two SLICELs. These two configurations alternate column by column across the FPGA. SLICELs (L=logic) can only be used for logic implementation. SLICEMs (M=memory) can implement either logic or memory as needed. Each SLICE element contains four Look-Up Tables and eight Flip-Flops. CLBs are combined to create a "reconfiguration frame". A reconfiguration frame is the smallest amount of logic that can be partially reconfigured. The reconfiguration frames on the Virtex 5 consists of 20 CLBs.

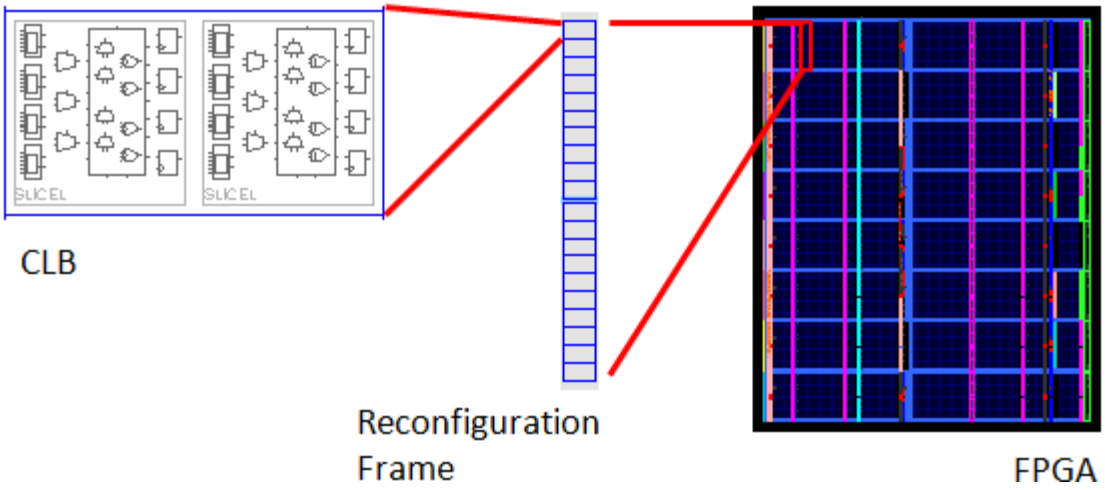

Figure 3.2 FPGA Resource Structure

Images taken from Xilinx PlanAhead

In order to demonstrate the procedure for calculating the size of the reconfigurable partition, the partition created for the Decoder project will be used as an example. The resource requirements data from the module data extraction is needed to begin the reconfigurable partition size calculation. This data is given in Table 3.4. It is clear from the table that module b1 requires the most resources and therefore dictates the size of the partition. The values used in the calculation are the SLICEL and SLICEM values. Note that the total number of SLICE elements required can be directly calculated from the Slice Reg and LUT requirements:

$$SLICE_{REQ} = 338\ LUT * \frac{1\ SLICE}{4\ LUTs} = 85 \text{ SLICEs} \quad (3.1)$$

The SLICEs are reported in a 1:1 ratio to be evenly distributed because both types can implement logic. However, the requirements will adjust to the distribution of the CLBs that are selected (i.e. if two columns are selected, the ratio will become 1:3, one

SLICEM to three SLICELs). Therefore, there is no need to consider the breakdown of the SLICE types.

Table 3.4 Reconfigurable Module Set Resource Requirements

Module Code	Module Name	Slice Reg	LUT	SLICEL	SLICEM
b1	bits2prm_ld8k	158	338	43	42
b3	CheckParityPitch	53	151	19	19

Because the height of a reconfigurable frame is fixed at 20 CLBs, a simple equation can be used to determine the minimum number of frames, *n*, necessary for the reconfigurable partition:

$$n = \left\lceil \frac{SLICEL+SLICEM}{40} \right\rceil \tag{3.2}$$

Inserting the values from Table 3.4 into equation 3.2, the minimum number of frames required for this reconfigurable partition is 3. However, this number is insufficient for a practical implementation. The reported resources required are only for the actual logic of a reconfigurable module and does not take into account space needed for signal routing. Attempting to implement the design with the minimum number of frames may cause the implementation to fail. Therefore, a simple method to account for this extra resource requirement would be to double the minimum value. Because the design size for this project was well below the capacity of the target FPGA, 10 frames were selected for the reconfigurable partition. However, this selection has a direct impact on the execution time of a partial reconfiguration. An analysis of this impact is given in section 4.2.

Once the reconfigurable partition has been defined, design configurations can be defined and implemented. A design configuration corresponds to a single orientation of

static logic and reconfigurable modules on the chip. For a design with a single reconfigurable partition containing three reconfigurable modules, only three design configurations exist, as shown in Figure 3.3. A design with two reconfigurable partitions containing three reconfigurable modules each, the number of design configurations increases to 9, as shown in Figure 3.4. However, it is unnecessary to define every possible combination. It is sufficient to define only as many configurations as it takes to ensure that every reconfigurable module is represented at least once.

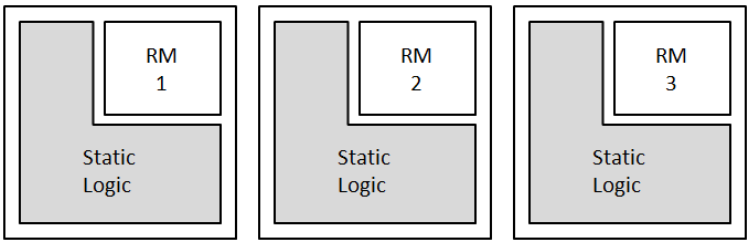

Figure 3.3 Design Configurations Example 1

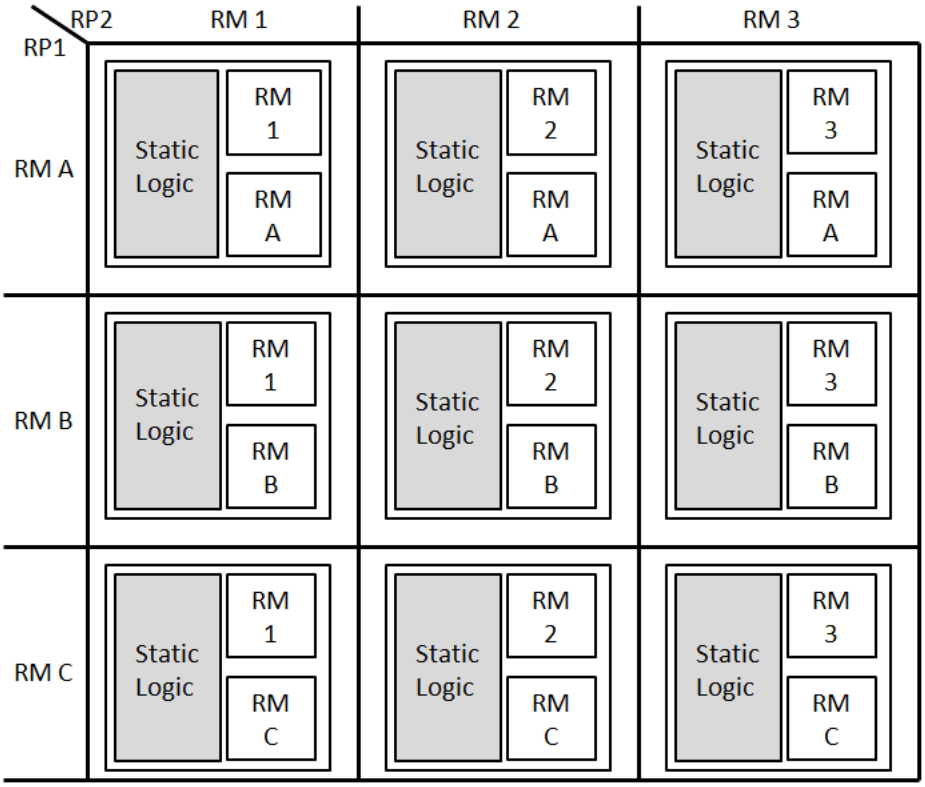

Figure 3.4 Design Configurations Example 2

After all design configurations are defined, they can be implemented. Implementation consists of three main stages: MAP, Place-and-Route, and bitfile generation. The MAP and Place-and-Route stages analyze the design and automatically map components and interconnections onto the FPGA. The bitfile generation stage creates multiple bitfiles for every design configuration. For each configuration, a full bitfile is created that represents the design with the designated reconfigurable modules already configured into the correct partitions. Also, partial bit files will be created for any reconfigurable modules defined in the configuration. Therefore, for this project, the Decoder used two different configurations: static plus RM b1 and static plus RM b3. This generated two full bitfiles and two partial bitfiles. Once the bitfiles are generated, the full

bitfiles can be used to program the FPGA in the same manner as a static project. The partial bitfiles can be loaded into system memory and then be used in conjunction with a Microblaze control program to dynamically reconfigure the reconfigurable partition on the FPGA.

3.3 Software Testing Platform

As stated previously, the goals of this testing application are to run the Decoder, manage the partial reconfiguration process, and perform timing analysis. The system is controlled by a Microblaze that communicates with the other peripherals through the Processor Local Bus (PLB). Attached to the PLB are the system memory, the ICAP, the external system timer, and the Decoder logic. The code for the testing program that is run on the Microblaze is given in APPENDIX C.

One of the major design decisions associated with testing the DPR system was how to communicate with the FPGA to deliver the partial bitfiles and frame data to the system. The first approach conceived was to hardcode the bitfiles into a BRAM using the Xilinx CORE Generator application. This, however, proved paradoxical because the .COE file needed to create the memory core could only be built after the partial bit files were created which in turn needed the memory core files to be generated. A second approach attempted was to use a separate C program to communicate with the FPGA over the UART. This technique was very time consuming, requiring multiple millions of read and writes every execution to transmit the two 59 kB files. Therefore, this option was ultimately abandoned. Finally, it was decided to use a third option that involved embedding the bitfile and frame data into the testing application as constant data arrays. Each bitfile was broken into 32-bit words and added to the program in a header file. From

there, at runtime, the data would be copied from the arrays into DRAM. This approach was a compromise required by the FPGA development board available for this work. In ordinary practice partial bit files are stored on a separate flash device and then copied into heap memory for use when reconfiguring.

After the bitfiles are loaded into memory, the program then begins the decode section. This section runs in a loop for a predetermined number of times. For each loop, a new set of frame data is loaded from memory. Then, a start signal is sent to the Decoder logic. While the Decoder is running, the program waits for a load signal to reconfigure the reconfigurable partition. When the signal is received, the program launches the Xilinx provided function that performs the necessary operations required to read a partial bitfile from memory into the HWICAP FIFO and then execute the partial reconfiguration of the reconfigurable partition. Finally, the program waits for the Decoder to report that it has completed a frame and finishes by printing out the timing data collected.

Three methods were proposed for collecting timing data for the Decoder. The first method utilizes the profiling tool provided in the Xilinx SDK. This tool uses an interrupt method that halts execution at regular intervals and determines which function the program is currently executing in. The tool counts how many times each function is identified and, after the program finishes, presents the data in a table including rows for each function and columns representing the number of calls to that function and the amount of time taken per call to run that function. However, after implementing this tool, it was determined that the results reported were too inconsistent to make a valid conclusion. Therefore, this method was discarded. The second approach uses the ChipScope debugging application. This approach requires the inclusion of a ChipScope

core in the hardware platform design. This core can be added in Xilinx PlanAhead prior to design implementation. Four signals are needed as triggers for timing: the application start signal, the application done signal, the load reconfigurable module signal, and the reconfigurable module ready signal. However, when the core was created for this project, the PlanAhead software assigned the start and done signals to the same trigger. This prevented the ChipScope Analyzer program from correctly timing the program. Therefore, this approach was also discarded. The final approach uses a hardware timer with a software interface to time the execution of the application. This approach uses an *xps_timer* core added to the design in the Xilinx EDK during the hardware platform creation. The *xps_timer* is accessed by the testing program that can start, stop reset, and get the current value of the timer. This method provided the most accurate measurements of the time taken by the Decoder. The timing data obtained by this method is presented and analyzed in the next chapter.

CHAPTER IV

RESULTS

One of the goals of this project was to determine if dynamic partial reconfiguration provides a suitable and possibly beneficial replacement for the static configuration of an HDL design. This section will present data and compare it to data collected from the static configuration. The results presented in this section were aggregated from different sources. The size data presented is taken from unit synthesis outlined in section 3.1, and the timing data was generated using a custom test application that is described in section 3.3.

4.1 Design Size

One of the obvious benefits for converting a project to use dynamic partial reconfiguration is that it will reduce the overall design size. This section presents size data and a quantitative analysis of the size benefits earned by implementing a dynamic partial reconfiguration system.

4.1.1 Size Data

Resource requirement data was collected for the 18 modules that are in the first instantiation level. For each module, data was recorded for the following five resources: SLICELs, SLICEMs, and DSP48E. This data is presented in Table 4.1. DSP48E is the component name of the DSP type resource found on the FPGA.

Table 4.1 RM Resource Data

Module Code	Module Name	SLICEL	SLICEM	DSP48E
b1	bits2prm_ld8k	43	42	0
b3	CheckParityPitch	19	19	0
b4	D_lsp	281	280	0
b13	int_qlpc	131	130	0
u1	copy	22	21	0
b16	Dec_lag3	49	48	0
b17	Pred_lt_3	75	75	0
b18	Random	50	50	1
b19	de_acelp	35	34	0
b20	Dec_gain	359	359	0
b24	syn_filt	81	81	0
b25	Weight_Az	30	29	0
b26	Residu	62	62	0
b27	calc_st_filt	187	186	0
b29	filt_mu	103	103	0
b30	scale_st	123	122	0
b31	pst_ltp	699	698	0
b36	post_process	115	115	0

4.1.2 Size Analysis

The size of the entire static logic configuration can be computed by summing all of the resources for each of the reconfigurable modules and adding that total to resources required for the static portion of the Decoder. This result is computed by adding rows 2 and 3 in Table 4.2 and shown in row 1 of Table 4.3. Using the full reconfigurable module set from the first instantiation level of the Decoder, the size of the Decoder's reconfigurable partition can be calculated by taking the resource requirements of the largest reconfigurable module, identified as RM b31 and given in row 4 of Table 4.2Table

4.1. Adding this value to the Decoder's static logic resources, row 2 of Table 4.2, will result in the design size required for the partial reconfiguration configuration of the design. This result is given in row 2 of Table 4.3. Row 3 of Table 4.3 shows that the partial reconfiguration design gives an over 50% reduction in SLICE requirements.

Table 4.2 Aggregated RM and Decoder Static Resource Requirements

	SLICEL	SLICEM	DSP48E
FPGA	12800	4480	64
Decoder Static	634	633	14
RM Sum	2464	2454	1
RM Maximum	699	698	1

Table 4.3 Static and Partial Reconfiguration Design Size Comparison

	SLICEL	SLICEM	DSP48E
Static Configuration	3098	3087	15
DPR Configuration	1333	1331	15
Percent Improvement	56.97%	56.88%	0.00%

A design size comparison can be created based on these values. Figure 4.1 shows this comparison as a percentage breakdown of usage of the FPGA. Compared to the static Decoder configuration, which requires 35.82% of the FPGA, the DPR configuration only requires 15.46% of the FPGA. This is a space saving of 20.36%, enough space to implement a second Decoder channel if desired. The results in this section prove that DPR does provide a size benefit over equivalent static systems.

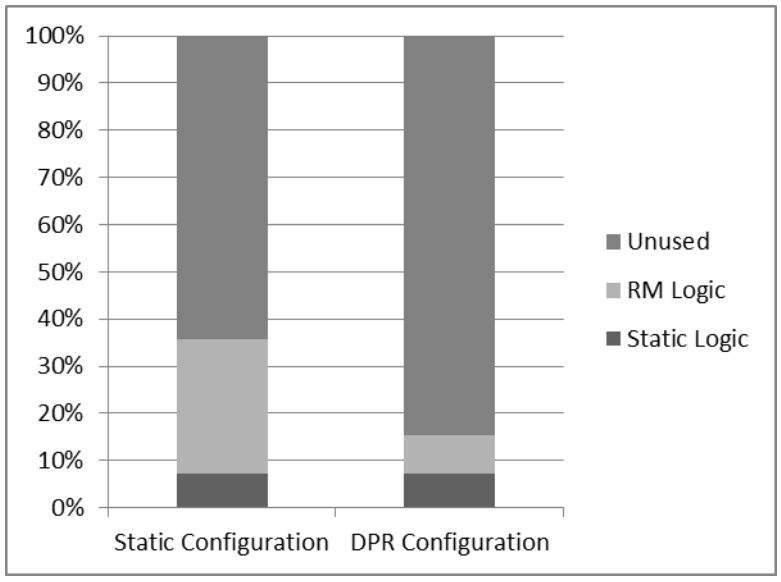

Figure 4.1 FPGA Usage Comparison

4.2 Application Runtime

4.2.1 Timing Data

Timing data was recorded for five seconds of audio data, equating to 500 frames. For each frame, 126 data points were recorded. These 126 points correspond to the following: the start and finish times of the Decoder, the start and finish times of the two partial reconfiguration operations, the start and finish times of the 15 HWICAP FIFO writes for each partial reconfiguration, and the start and finish times of the 15 HWICAP reconfiguration processes for each partial reconfiguration. Although not drawn to scale, Figure 4.2 shows where each of these points occurs during the execution of the Decoder.

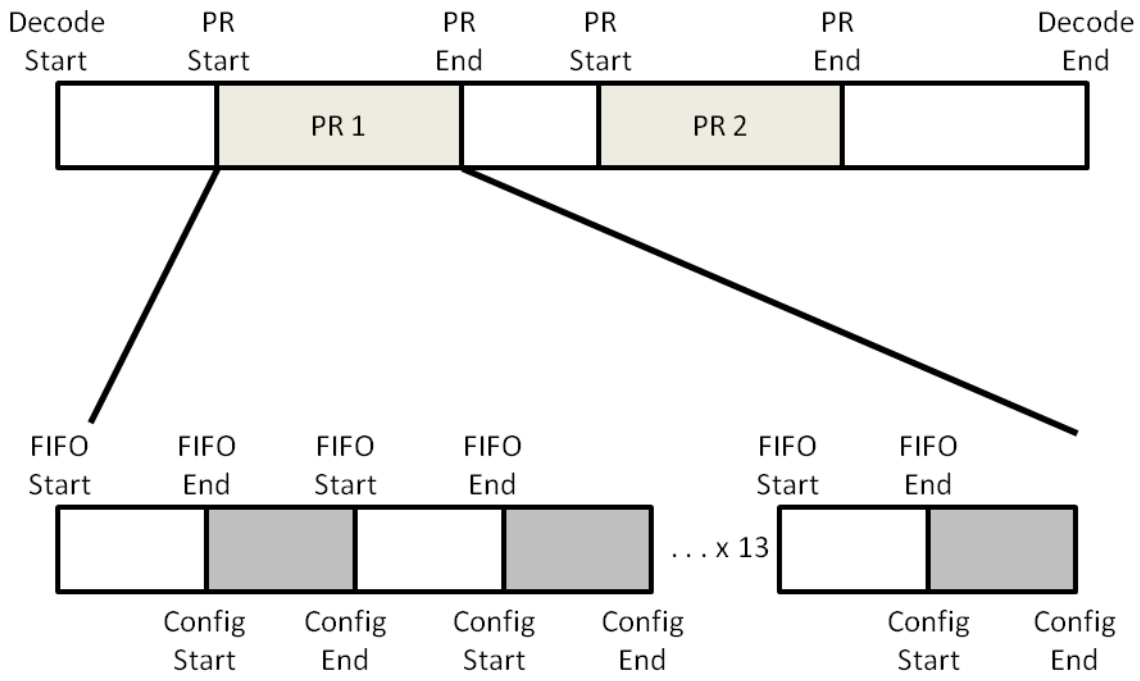

Figure 4.2 Timing Data Points

A summary of the data collected is presented in Table 4.4. The data in the table is presented in milliseconds. The Decoder column represents the total time taken to decode a single frame of audio. The second column, “Partial Reconfig”, gives the time required to perform a single partial reconfiguration. The “FIFO Write (FULL)” column corresponds to the amount of time necessary to fill the entire HWICAP FIFO. The “FIFO Config (FULL)” column gives the time taken to configure the partition when the FIFO is full. The “FIFO Write (PARTIAL)” and “FIFO Config (PARTIAL)” columns are similar to the previous two, respectively, but correspond to the runtime of the final loop when the FIFO is not filled completely. The “Decode” column is summarized from a sample of 500 because it is only recorded once per frame. The “Partial Reconfig” time is recorded twice per frame; therefore, 1000 samples were available for analysis. This is also true for the

two “Partial FIFO” columns. Finally, the “Full FIFO” columns are recorded 14 times per partial reconfiguration and therefore 28 times per loop, which calculates to 14,000 samples over 500 runs.

Table 4.4 Timing Data for 1 Frame of Audio

	Decode[1]	Partial Reconfig[2]	FIFO Write[3] (FULL)	FIFO Config[3] (FULL)	FIFO Write[2] (PARTIAL)	FIFO Config[2] (PARTIAL)
Average Time (ms)	203.49	101.13	6.8602	0.02787	4.1714	0.02403
Standard Deviation	0.00282	0.00039	0.00009	0.00008	0.0001	0.00007

[1] 500 Samples
[2] 1000 Samples
[3] 14000 Samples

4.2.2 Timing Analysis

4.2.2.1 Test System Analysis

A frame decode timeline can be constructed as shown in Figure 4.3. Note that this figure is not drawn to scale. Figure 4.4 illustrates what percentage of the runtime is spent in each of the execution phases. As seen in the figure, the amount of time taken by the partial reconfiguration process greatly outweighs the time spent in the Decoder logic.

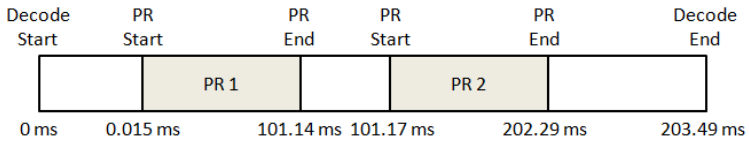

Figure 4.3 Frame Decode Timeline

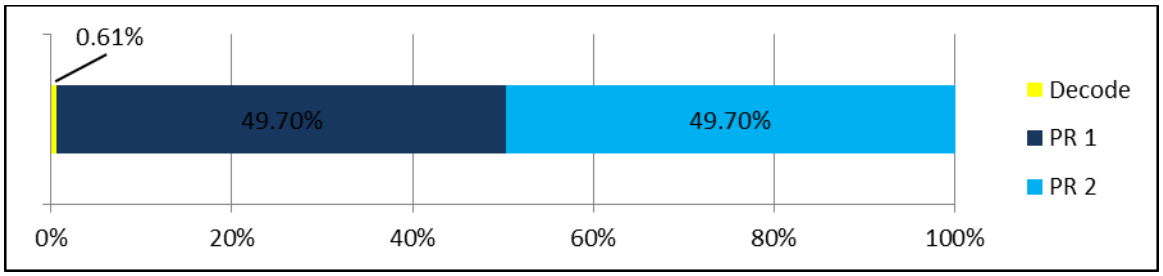

Figure 4.4 Frame Decode Percent Contributions

By subtracting the partial reconfiguration runtimes from the total runtime, it is determined that the actual decoding logic only runs for 1.23 milliseconds. Testing by Duplantis et al. showed that the Decoder running at 50 MHz completed a frame in approximately 2.6 milliseconds [8]. For this project, the Decoder was run at 100 MHz; therefore the observed runtime is in the expected range.

Conversely, the partial reconfiguration process did not execute in the expected timeframe. According to Table 6-1 in [17], the HWICAP has a bandwidth of 3.2 Gbps (32-bit data width at 100 MHz). Assuming this throughput, the HWICAP will be bounded by the FIFO speed. However, with a bitfile consisting of 14943 double words, a FIFO write time of 10 clock cycles, and a PLB running at 100 MHz, a pessimistic calculation of the reconfiguration time would only require 1.5 milliseconds as shown in (4.1).

$$PR_{Time} = 14943\ writes * 10\ cycles * \frac{10ns}{cycle} = 1.4943ms \quad (4.1)$$

An investigation of the *DeviceWrite* function shows that the FIFO is filled before launching the *XHwIcap_StartConfig* function, which then runs to completion before returning to the beginning of the loop to refill the FIFO if necessary. This method of execution effectively removes any benefits gained by using the FIFO structure.

Furthermore, as shown in Figure 4.5, the FIFO write is the dominating value in the partial reconfiguration runtime.

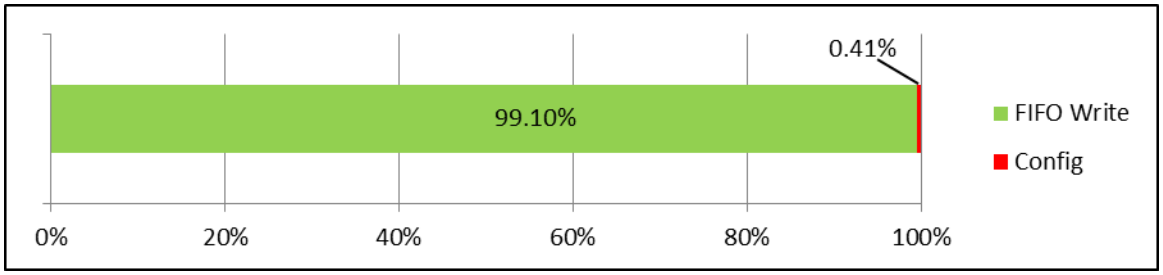

Figure 4.5 PR Percent Contributions

This calculation presents a further question as to why the FIFO write process is taking 6.86 milliseconds to fill the FIFO. Knowing that the FIFO has a depth of 1024, equation (4.2) shows the calculation of the number of clock cycles necessary to write a double word to the FIFO. It is undetermined what is causing the multiple orders of magnitude increase in the number of cycles required to write data to the FIFO.

$$WRITE_{FIFO} = \frac{6.86\ ms}{1024\ writes} * \frac{1\ cycle}{.000001\ ms} = \frac{6700\ cycles}{write} \quad (4.2)$$

4.2.2.2 Timing Analysis Extrapolation

Moreover, the timing data presented above only reflects the dynamic partial reconfiguration system implementing the first two reconfigurable modules. An extrapolation of the timing data, combined with the size data from the previous section, can show the estimated timing results for a reconfigurable module set that includes every module in the first instantiation level. The size data recorded for this extrapolated system indicates that the required reconfigurable partition would have to be at least four times

the size of the reconfigurable partition used in the project, which results in a minimum partial bitfile size of 60,000 double words. Equation (4.1) can be used to calculate the new runtime required to perform a partial reconfiguration, as shown in (4.3).

$$PR_{TIME} = 60000\ writes * 6700\ cycles * \frac{10ns}{cycle} = 4.02s \quad (4.3)$$

In the average case, the extrapolated decode process will require 31 reconfigurations per frame. Equation (4.4) shows the calculation of the Decoder runtime with the extrapolated system design.

$$Decode\ Runtime = 1.23\ ms + (31 * 4.02\ s) = 124.62\ s \quad (4.4)$$

It should be noted that even an estimated best case FIFO write time will exceed the 10ms period for the full reconfigurable module set, as shown in (4.5) and (4.6).

$$PR_{TIME_{optimal}} = 60000\ writes * 1\ cycle * \frac{10ns}{cycle} = 0.6ms \quad (4.5)$$

$$Decode\ Runtime_{optimal} = 1.23\ ms + (31 * 0.6\ ms) = 19.83\ ms \quad (4.6)$$

4.2.2.3 PR Application Period

As shown in the previous two sections, the standard Xilinx dynamic partial reconfiguration flow is insufficient to support the high frequency/small period timing requirements of the G.729 Voice Decoder Application even in a reduced capacity (2 reconfigurable modules versus 18 reconfigurable modules). Therefore, it is clear that there is a minimum application period that is supportable by this flow. This minimum period can be calculated using a generalized form of equation (4.4) presented below in (4.7). In (4.7), the value of t_{pr} represents the time required to reconfigure the reconfigurable partition and can be calculated by substituting the size of the partial bitfile

for the reconfigurable module into (4.1). The n_{pr} element corresponds to the number of partial reconfigurations that will occur every period of the application. Finally, the t_{logic} element accounts for the time necessary to perform the logic operations for the application. Using (4.7), application designers will be able to determine if their application is suitable for dynamic partial reconfiguration by checking if the application's frequency is greater than the T_{DPR}.

$$T_{DPR_{MIN}} = (t_{pr} * n_{pr}) + t_{logic} \tag{4.7}$$

CHAPTER V

FUTURE WORK AND CONCLUSIONS

5.1 Conclusions

Dynamic partial reconfiguration of FPGA's is becoming a very popular area of research. Dynamic partial reconfiguration allows designers to create smaller and more flexible systems that are critical in extreme environments. This main contribution of this project was to assess the ability of Xilinx's standard partial reconfiguration flow to provide a useable Commercial Off-The-Shelf solution for converting static HDL designs to dynamically, partially reconfigurable designs.

A first component of this assessment was to analyze the design changes required to convert an application from static to partially reconfigurable. In this assessment, multiple approaches were presented for selecting the reconfigurable module set to be used for the final system. Furthermore, three different methods of abstracting the port lists for the modules in the reconfigurable module set were presented. Also, a listing of the architecture modifications necessary was enumerated.

Secondly, this project gives an overview of the process necessary to design and implement a hardware platform for supporting a dynamic partial reconfiguration application and a software application to run it. In this process, a method for calculating the minimum reconfigurable partition size required is presented. An in depth tutorial of this process is provided in APPENDIX B.

A final contribution of this project was to provide an examination of software verification techniques for dynamic partial reconfiguration systems, empirical timing and design-size data, and analysis that can be used by future designers to predict how conducive an application is for conversion to a dynamic partial reconfiguration system. This data proved that dynamic partial reconfiguration provides a substantial reduction in design size for the Decoder application. However, the timing data showed that the standard implementation provided by Xilinx is not sufficient to support the 10ms period of the Decoder. Therefore, a minimum partial reconfiguration period equation is presented to allow designers to predict whether a given application will be able to support dynamic partial reconfiguration.

5.2 Future Work

As evident by the timing analysis in section 4.2.2, work must be done to improve the speed at which data can be transferred to the ICAP. As stated in the introduction, research has been done in this area, as in [12]; however, this was a custom design. Because this project assesses the tools and implementations provided by Xilinx, future work must be done by Xilinx to provide a more efficient method for bitfile data communication. One suggested approach would be to utilize the FIFO already in place by reconfiguring data as it is loaded into the FIFO.

Another area of future work that was mentioned in previous sections is the possibility of creating a partial reconfiguration region hierarchy. This hierarchy would allow regions inside of a reconfigurable partition to be reconfigured independently of the parent region. This design would be useful for systems such as the Decoder that have multiple instantiation levels. This would further increase the size benefits of using

dynamic partial reconfiguration, but would also incur further costs to runtime due to the increased number of reconfigurations.

A final area of research that would be useful for the dynamic partial reconfiguration flow would be to create template HDL files that would allow designers to easily communicate with the ICAP without the need for a software interface. The removal of the software interface could greatly increase the speed at which partitions could be reconfigured. Furthermore, this would allow programmers to create deployment ready systems without the necessity for custom HDL designs.

REFERENCES

[1] R.J. Petersen and B.L. Hutchings, "An Assessment of the Suitability of FPGA-Based Systems for use in Digital Signal Processing," in *5th International Workshop on Field-Programmable Logic and Applications*, Oxford, England, 1995.

[2] M. Kristan, B. Loveland, and R. Sazanowicz. (2001) Dynamic Partial Reconfiguration of a Field Programmable Gate Array [Online]. Available: http://www.wpi.edu/Pubs/E-project/Available/E-project-030307-144157/unrestricted/MQP_GDC07_FINAL.pdf

[3] W. Lie and W. Feng-yan, "Dynamic partial reconfiguration in FPGAs," in *Third International Symposium on Intelligent Information Technology Application*, Nanchang, China, 2009.

[4] R. Tessier and W. Burleson. "Reconfigurable Computing for Digital Signal Processing: A Survey." *Journal of VLSI Signal Processing*, vol. 28, no. 1-2, pp. 7-27, 2001.

[5] J. Villasenor and B. Hutchings, "The Flexibility of Configurable Computing," *IEEE Signal Processing Magazine*, Sept. 1998, pp. 67–84.

[6] P. Sedcole et al. "Modular dynamic reconfiguration in Virtex FPGAs." *Computers and Digital Techniques, IEE Proceedings*, vol. 153, no. 3, pp. 157-164, 2006.

[7] B. Osterloh et al., "Dynamic Partial Reconfiguration in Space Applications," in *NASA/ESA Conference on Adaptive Hardware and Systems*, San Francisco, CA, 2009.

[8] S. Owens and J.Z. Thornton. (2010, December 1). Design Document for ITU G.729 FPGA Implementation [Online]. Available: http://www.ece.msstate.edu/courses/design/2010/codec/Final_Design_Document.pdf

[9] T. Huguet et al. (2011, December 2). Design Document for ITU G.729 FPGA Implementation [Online]. Available: http://www.ece.msstate.edu/courses/design/2011/codec2/Deliverables/FinalDocument2.pdf

[10] D. Mudd and J. Humphreys. (2012, 25 April). Design Document for Voice Decoder [Online]. Available: http://www.ece.msstate.edu/courses/design/2011/team_humphreys/deliverables/Final_Doc.pdf

[11] C. Duplantis et al. (2012, April 25). Design Document for ITU G.729 Voice Decoder [Online]. Available: http://www.ece.msstate.edu/courses/design/2012/team_ratcliff/documents/FinalDoc_Final.pdf

[12] P. Manet et al. "An Evaluation of Dynamic Partial Reconfiguration for Signal and Image Processing in Professional Electronics Applications". *EURASIP Journal on Embedded Systems*, vol. 2008, pp. 1-11, 2008.

[13] Claus et al. "Towards Rapid Dynamic Partial Reconfiguration in Video-Based Driver Assistance Systems". In Reconfigurable Computing: Architectures, Tools and Applications (ARCS), volume 5992 of LNCS, pp. 55-67. Springer, 2010.

[14] E. J. McDonald, "Runtime FPGA Partial Reconfiguration," in *IEEE Aerospace Conference*, Big Sky, MT, 2008.

[15] S. Bhandari et al. "Internal dynamic partial reconfiguration for real time signal processing on FPGA". *Indian Journal of Science and Technology*, vol. 3, no. 4, pp. 365-368, 2010.

[16] PlanAhead User Guide. Xilinx, UG632, 2010.

[17] Partial Reconfiguration User Guide. Xilinx, UG702, 2010.

[18] PlanAhead Software Tutorial Overview of the Partial Reconfiguration Flow. Xilinx, UG743, 2010.

[19] Lab 1: Intro to Partial Reconfiguration Flow Lab [Online]. Available: http://www.xilinx.com/university/workshops/partial-reconfiguration-flow/

[20] Lab 2: Creating Constraints and Performing Timing Analysis in a PR Design consists of HDL and Netlist Files Lab [Online]. Available: http://www.xilinx.com/university/workshops/partial-reconfiguration-flow/

[21] Adding EDK IP to an Embedded System Lab [Online]. Available: http://www.cosmiac.org/edk.html

[22] Adding Custom IP to an Embedded System Lab [Online]. Available: http://www.cosmiac.org/edk.html

[23] G.729: Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP). ITU-T, Rec. ITU-T G.729, 2012.

[24] http://www.xilinx.com/products/boards-and-kits/XUPV5-LX110T.htm

[25] S. Owens et al., "A Multi-Team Multi-Semester Large-Scale Capstone Project Experience," in *Proceedings of the ASEE Southeastern Section Annual Conference*, Starkville, MS, 2012.

APPENDIX A

G.729 DECODER TOP-LEVEL FSM

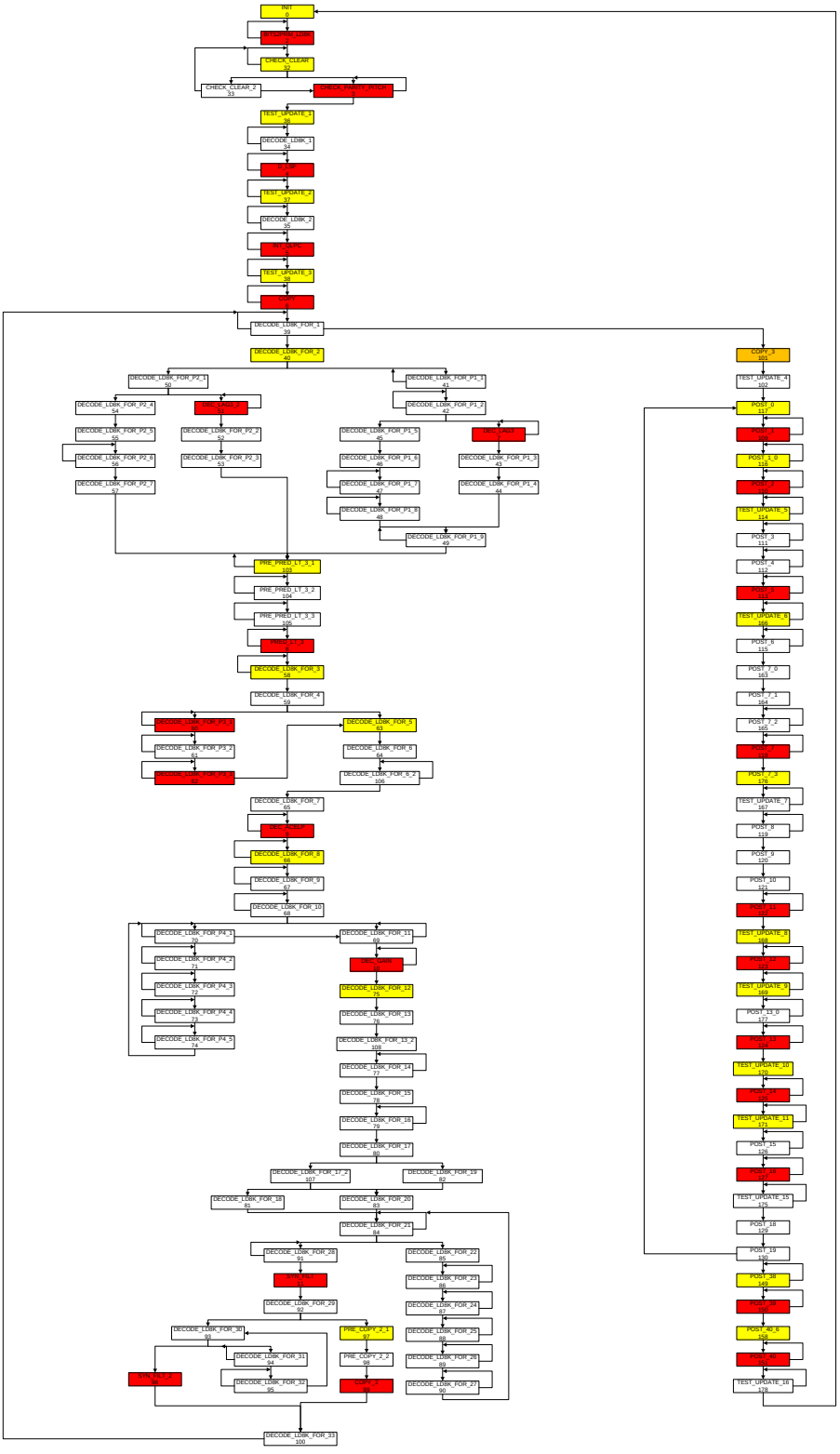

Figure A.1 Decoder Control FSM Overview

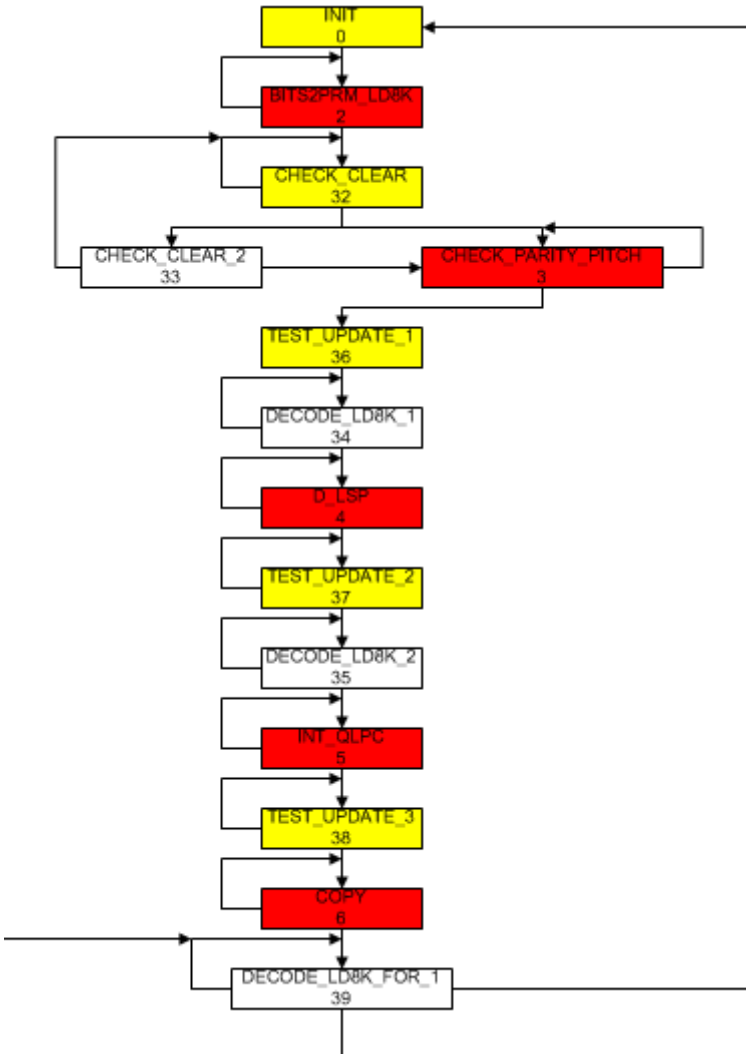

Figure A.2 Decoder Control FSM Detailed Part 1

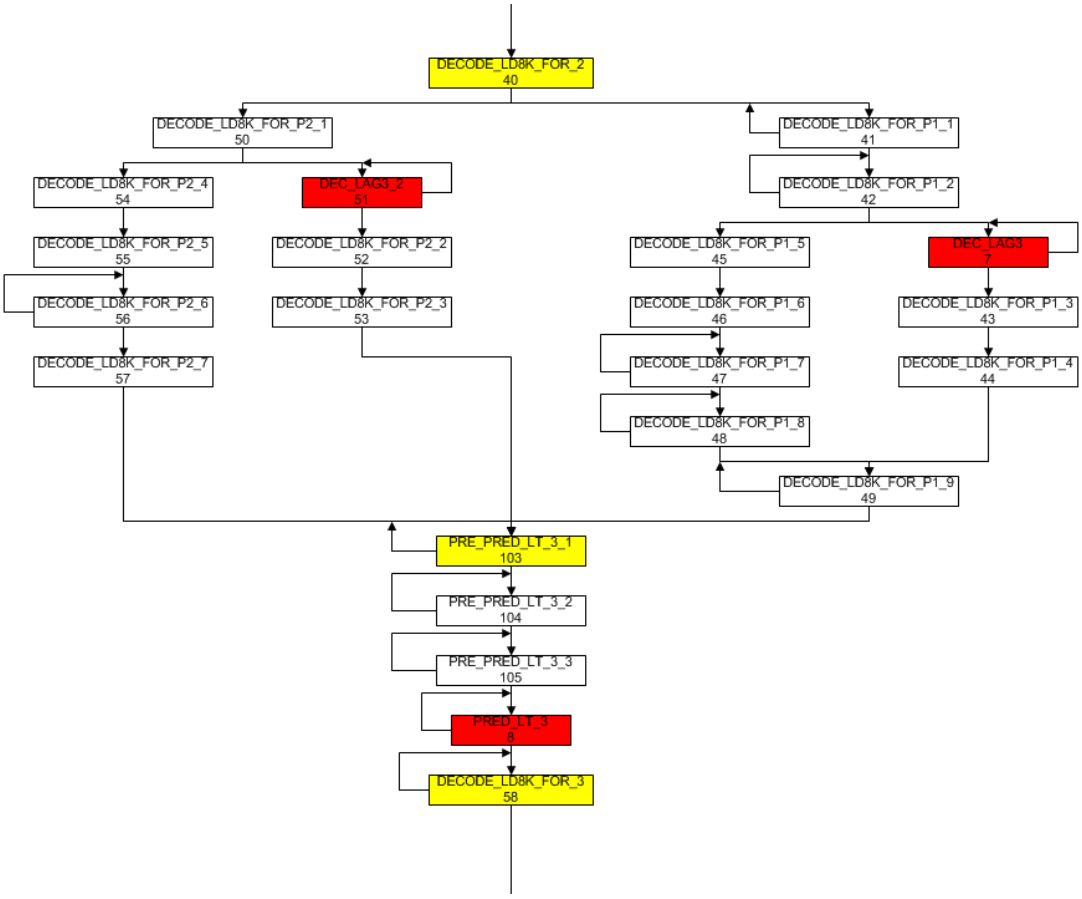

Figure A.3 Decoder Control FSM Detailed Part 2

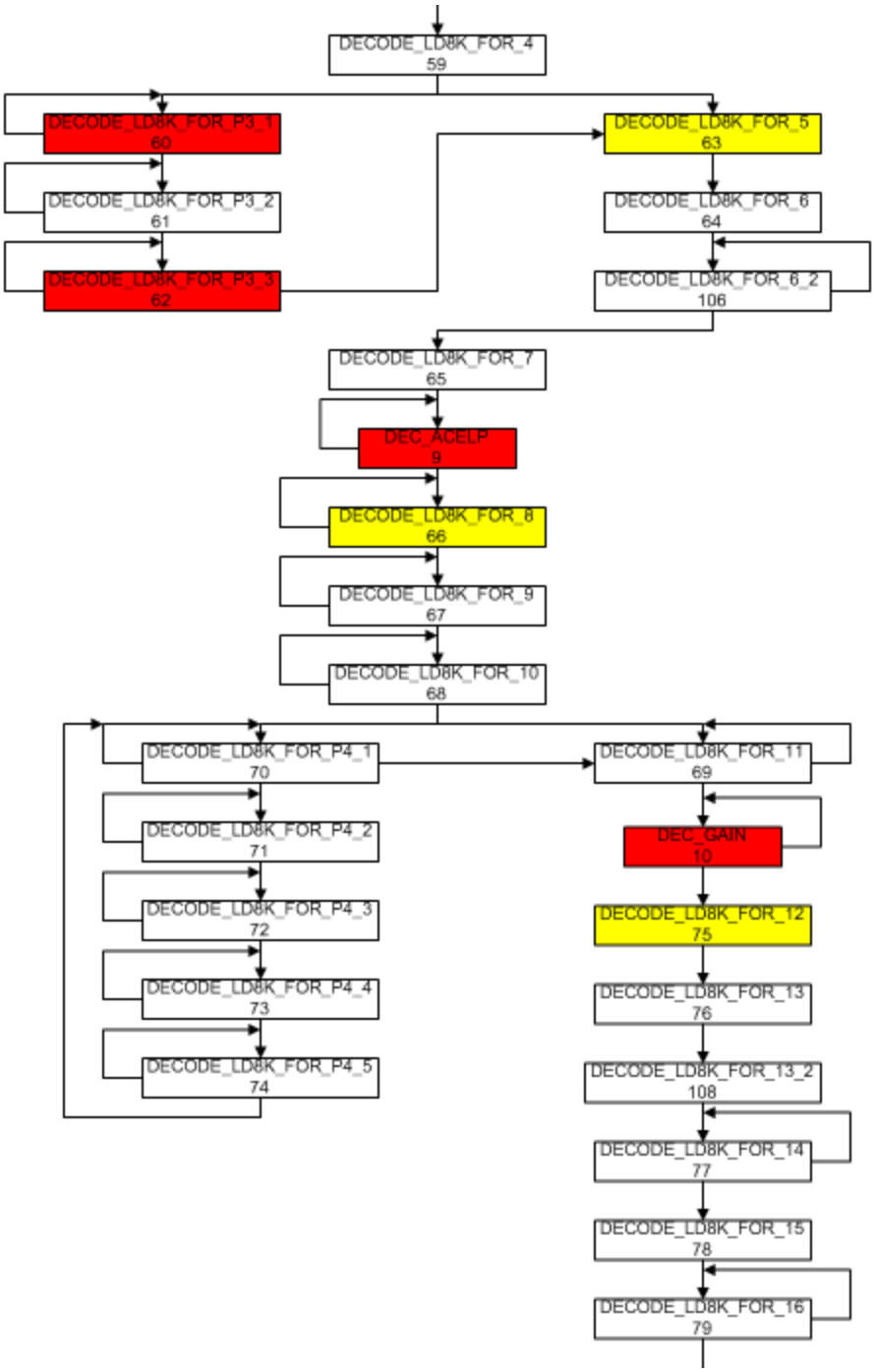

Figure A.4 Decoder Control FSM Detailed Part 3

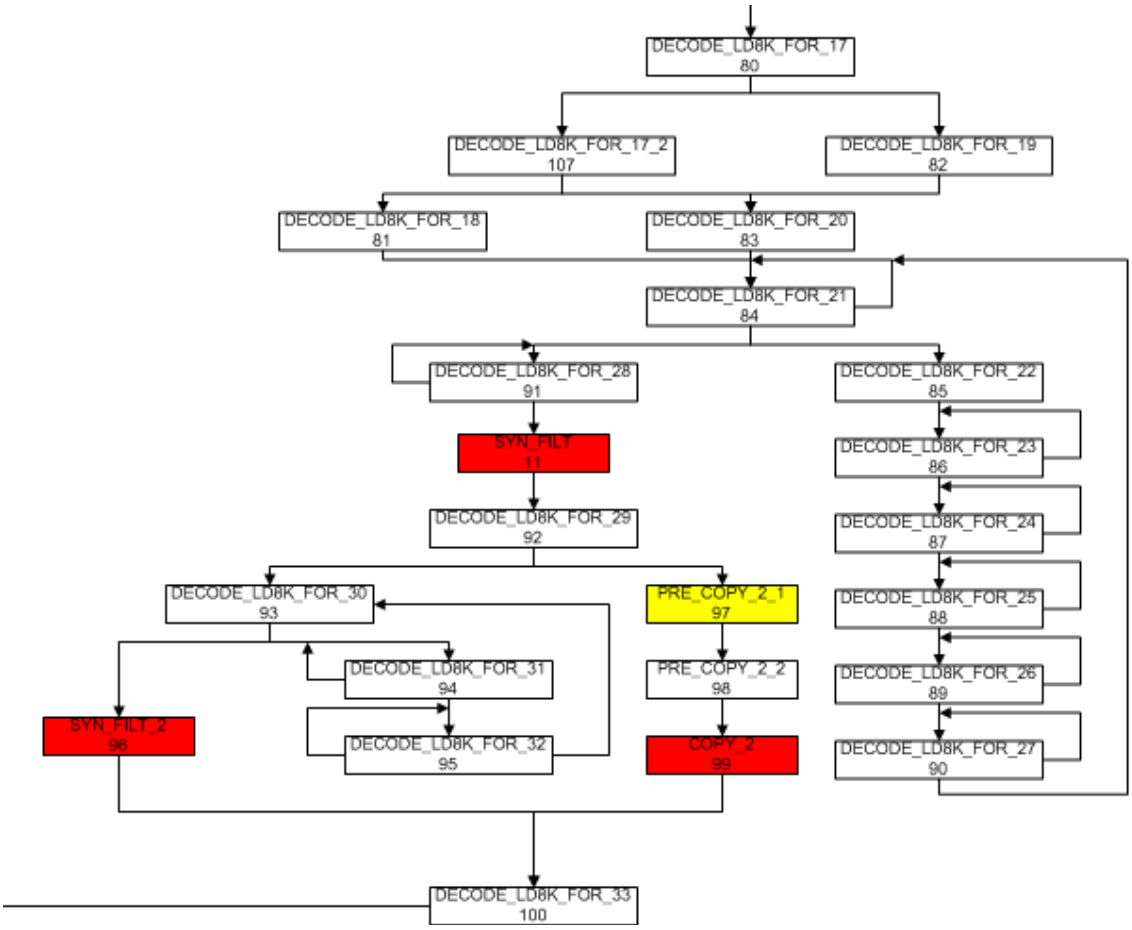

Figure A.5 Decoder Control FSM Detailed Part 4

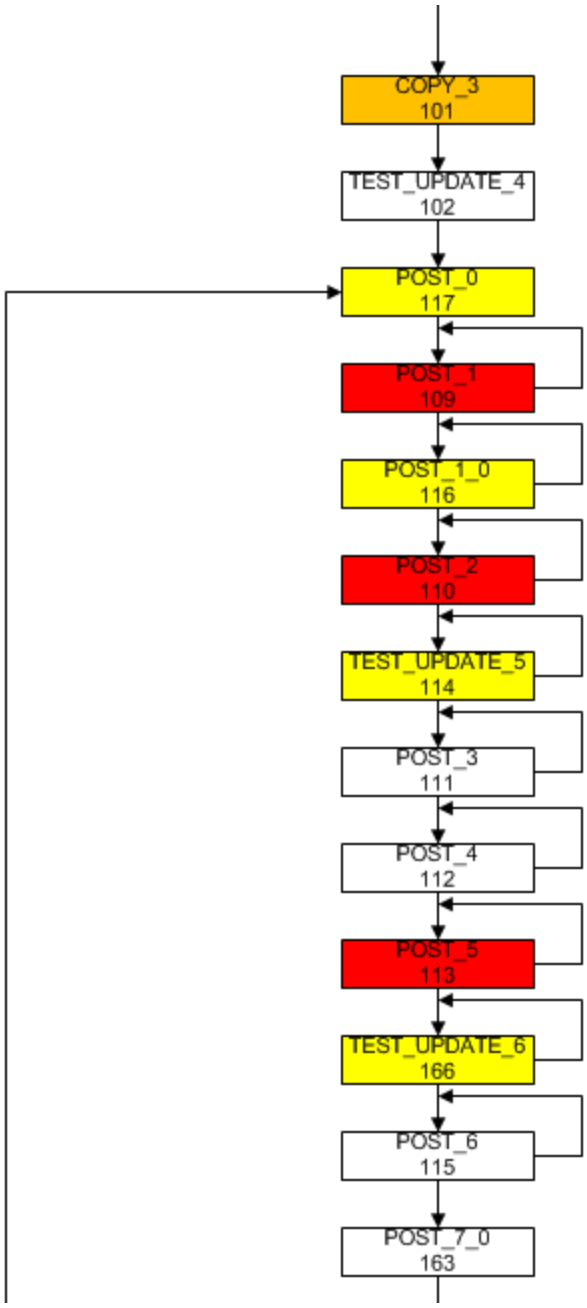

Figure A.6 Decoder Control FSM Detailed Part 5

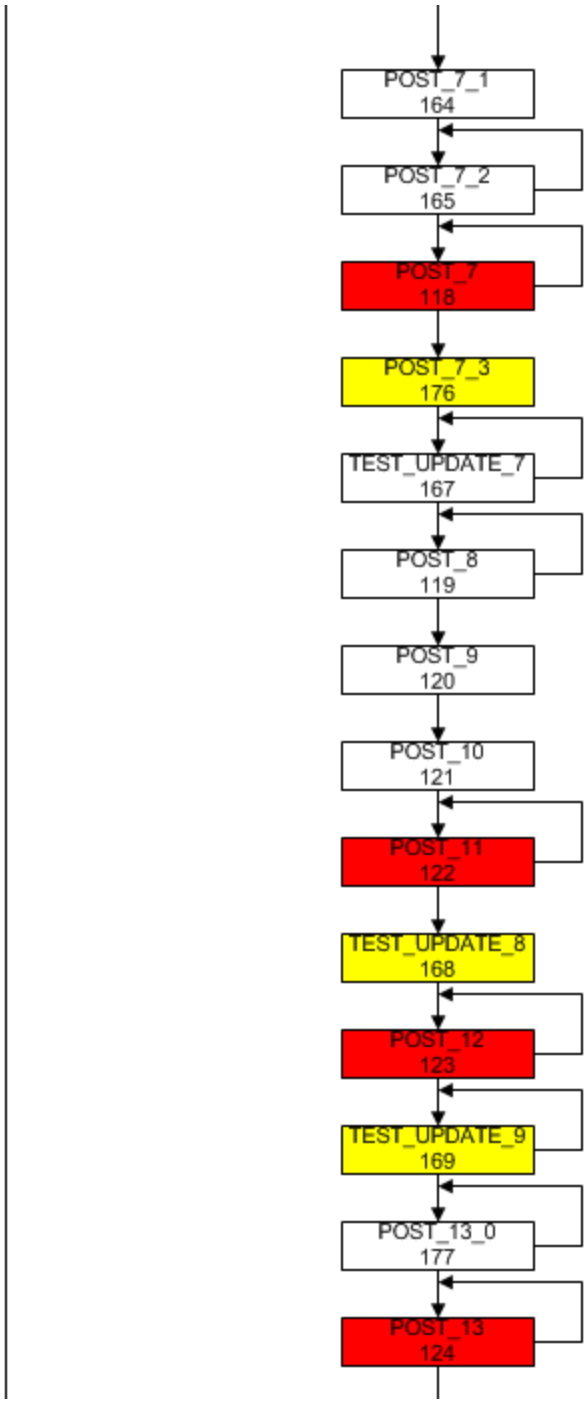

Figure A.7 Decoder Control FSM Detailed Part 6

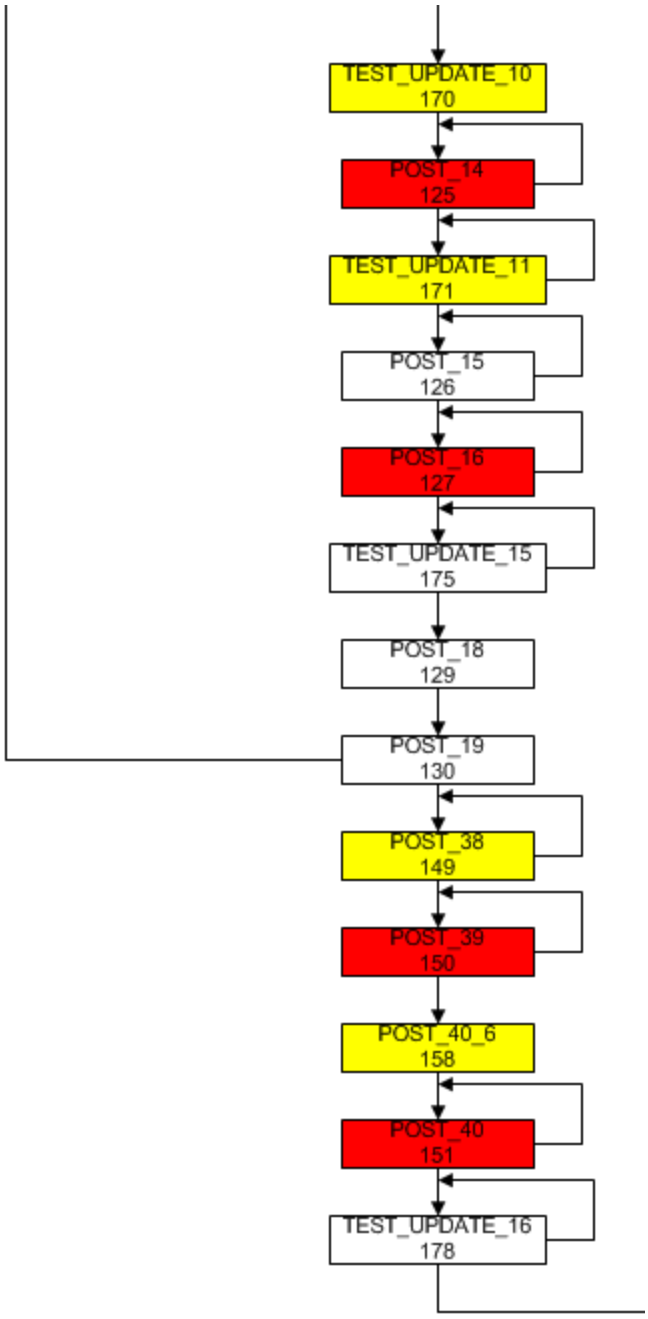

Figure A.8 Decoder Control FSM Detailed Part 7

APPENDIX B

DYNAMIC PARTIAL RECONFIGURATION HARDWARE PLATFORM DEVELOPMENT TUTORIAL

Dynamic partial reconfiguration development requires multiple Xilinx IDE's. The basic software flow is shown in Figure B.1. The first step is to create a project in Xilinx ISE. From here, the Microblaze system is created and designed in the Xilinx XPS/EDK tool. The next step is to configure the custom logic peripheral in a new instance of Xilinx ISE. After the custom logic peripheral is re-imported back into the XPS platform, the system can be synthesized in the original ISE window. The next step is to import the system netlist into the Xilinx PlanAhead software for partial reconfiguration design. In the PlanAhead tool, the reconfigurable partition is defined and the reconfigurable modules are associated with it. Then, the system is implemented and the bitfiles generated. With the full and partial bitfiles generated, the FPGA can be programmed with the Xilinx iMPACT tool, and the Xilinx SDK can be used to develop a test application for the system.

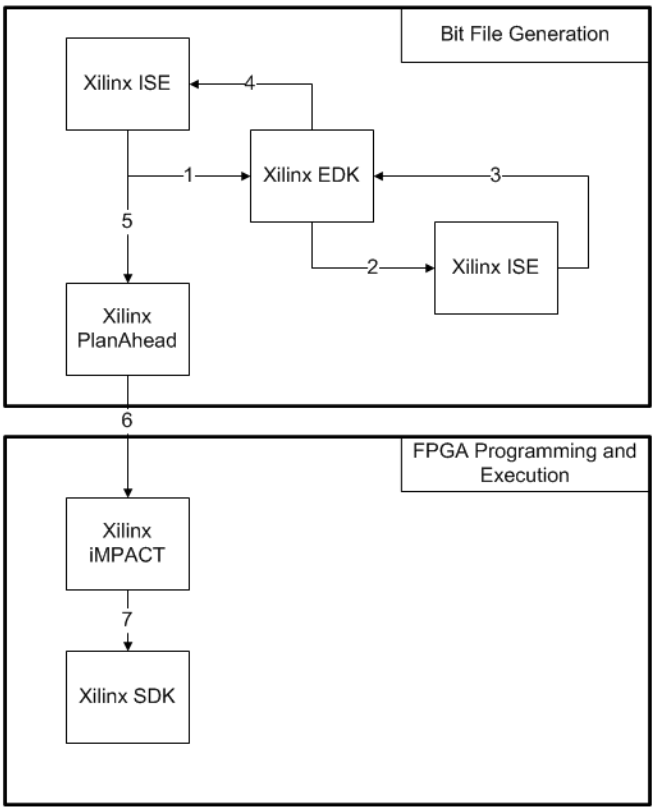

Figure B.1 Dynamic Partial Reconfiguration Software Flow Overview

This chapter details the process of developing a DPR project and executing it on a Xilinx FPGA. This process was developed using multiple tutorials provided by Xilinx [16]-[22].

B.1 File Structure

During the development cycle many files will be generated and passed from one software tool to another. Furthermore, partial bit file creation requires the use of multiple .ngc files that must have the same name. Therefore, an important step in developing a DPR project is setting up the file tree structure. The file structure used for this project is illustrated in Figure B.2.

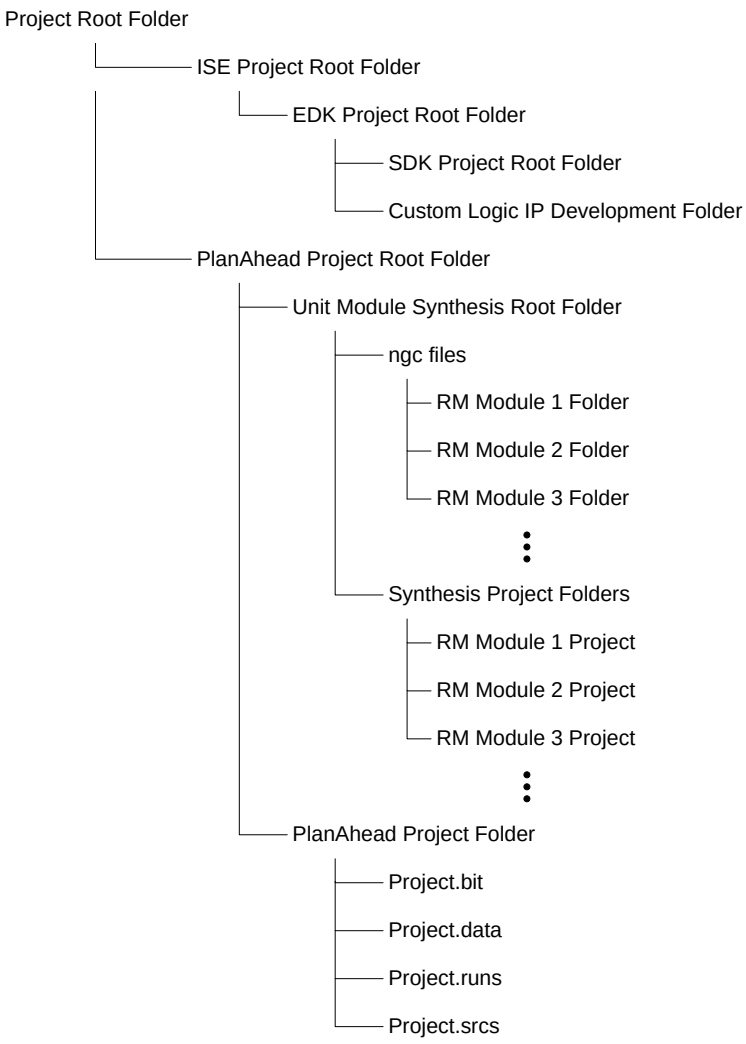

Figure B.2 DPR File Tree Structure

The figure above shows that there are two main folders inside the root project folder. The ISE Project Root Folder holds all files related to the platform development project including ISE, EDK, and SDK files. Inside this folder are all of the ISE project files and an EDK Project folder that is named based on the processor source created in the ISE project. The EDK Project folder contains all of the platform creation files, as well as the SDK project Folder and Custom Logic IP folder. The SDK project folder is where

all of the files associated with software development for application execution are stored. The Custom Logic IP folder is where all of the files associated with generating a custom logic peripheral to be added to the platform design in the EDK.

The second folder found in the Project root folder is the PlanAhead Root Folder. This folder holds all files associated with the PlanAhead IDE and partial bit file generation process. The first subfolder is the unit synthesis folder. This folder is used to hold all files related to .ngc file generation for reconfigurable modules. Due to Xilinx's common name constraint, a folder is needed for each reconfigurable module to identify which module the contained .ngc corresponds to. The other folder in this location contains the the project folders for each RM's ISE project used to generate the .ngc files. The second file in the PlanAhead root folder is the PlanAhead project folder that contains the PlanAhead project files and generated partial bit files. The bit files are generated and automatically stored in the Project.runs folder but are copied into the user created Project.bit folder for ease of access.

B.2 ISE - Project Creation

Xilinx ISE is a project manager that allows the user to build a system from scratch and then automatically launch other Xilinx IDE's when necessary. All of the development projects presented in this document were built in Xilinx ISE 12.4 64-bit environment.

The first step is to open the Xilinx ISE Project Navigator and launch the "New Project Wizard" found in "File -> New Project". The user can then input a project name and directory as shown in Figure B.3. The Top-level source type should be set to HDL.

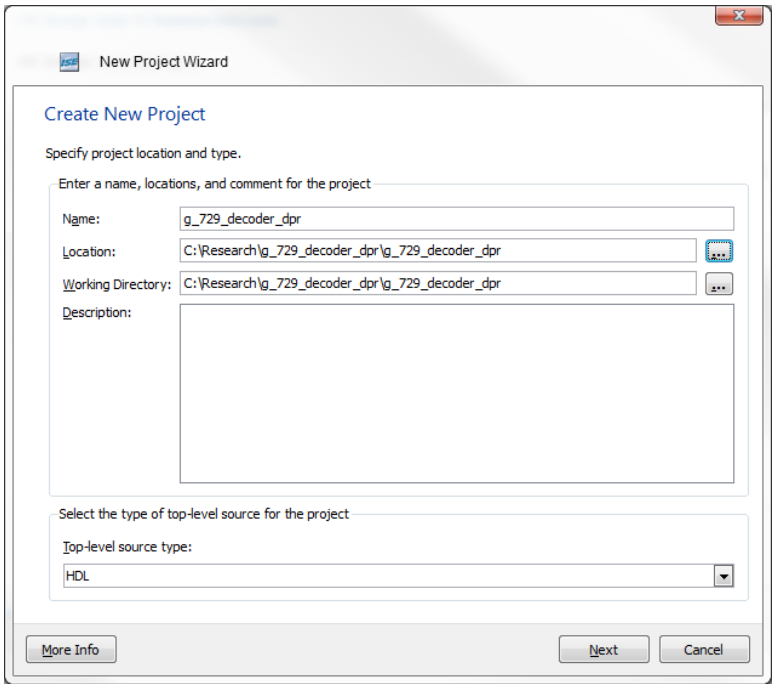

Figure B.3 ISE Create New Project Window

After proceeding to the next page, the user can set their hardware properties and preferred language. This project is designed to be executed on Xilinx's XUPV5-LX110T Development Platform which uses a Virtex-5 XC5VLX110T FPGA. This page is shown in Figure B.4.

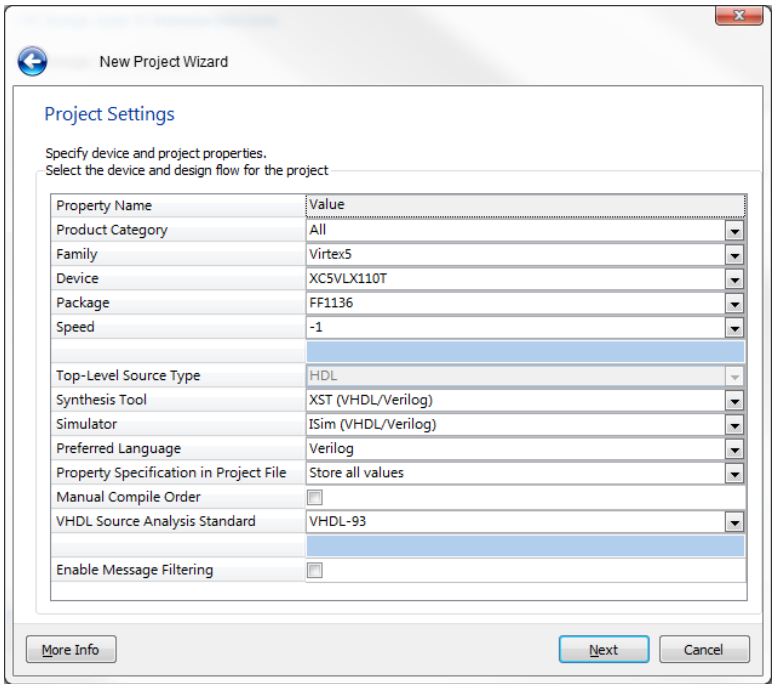

Figure B.4 ISE Project Settings Window

The final page allows the user to review the project and finish project creation. Once the project has been created, the user can add an embedded processor to the project through “Project -> New Source”. In the New Source Wizard window, the embedded processor option is selected and given a name and directory as demonstrated in Figure B.5.

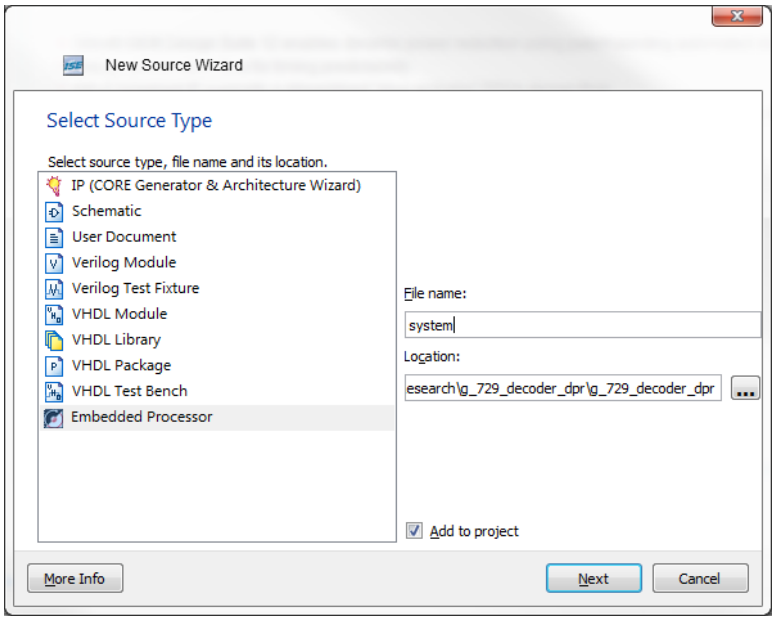

Figure B.5 ISE Select Source Type Window

When the embedded processor source is created, Xilinx ISE will automatically launch the EDK to customize the processor design. Once the EDK is open, it will detect that the loaded project does not have a design associated with it and will prompt the user to launch the Base System Builder (BSB) Wizard. The reference designs for the development board used in this project are not included in the standard Xilinx Suite installation; these files must be downloaded and associated with the EDK before the BSB Wizard can be run. Therefore, the option to run the BSB Wizard must be declined. The reference designs for the development board were downloaded from Xilinx's website [24]. The files are associated with the EDK by going to the Preferences window at "Edit -> Preferences -> Global Peripheral Repository Search Path" and setting the field to "../EDK-XUPV5-LX110T-Pack/lib". This process is shown in Figure B.6.

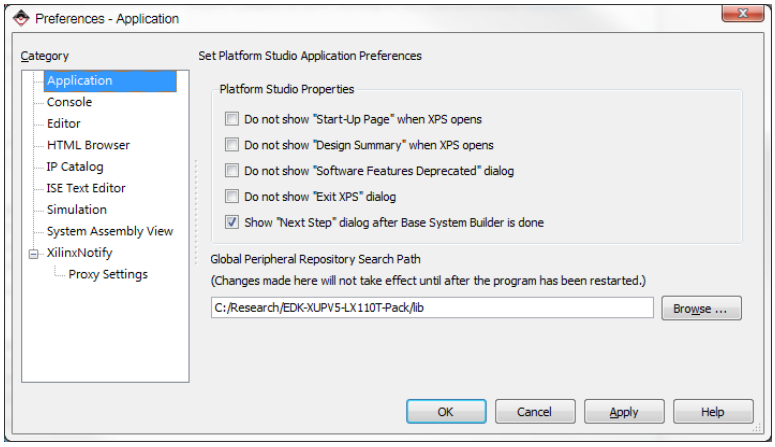

Figure B.6 XPS Application Preferences Window

In order for the EDK to associate with the new folder, it must be restarted. To do this, exit the EDK and return to the ISE Project Navigator. Select the embedded processor and launch "Manage Processor Design" in the Design tab under Design Utilities.

B.3 XPS - System Design I

The Embedded Development Kit is included in the Xilinx Platform Studio. It allows users to customize a virtual processor core by adding and modifying different peripherals such as memory, communication controllers, and user created peripherals. The EDK is used in this project to create a *Microblaze* processor to handle reconfiguration and data flow processes and to add a custom logic peripheral to the design. Xilinx XPS version 12.4 (nt64) was used for all development on this project.

B.3.1 Base System Builder

The first step to creating the processor design is to run the "BSB Wizard". The EDK will automatically launch the BSB for an empty project. The first screen is the *Board Selection* window. If the repository was linked successfully, the EDK should

automatically load the XUPV5-LX110T Evaluation Platform from the device data entered into the ISE project. The board screen is shown in Figure B.7.

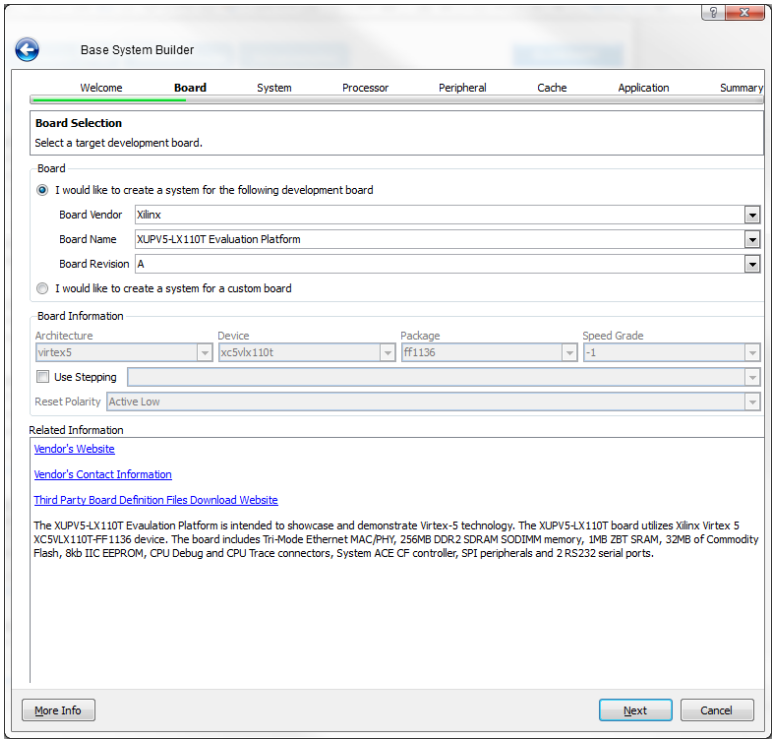

Figure B.7 XPS BSB Board Selection Window

The next screen is the *System Configuration* page. This page allows the user to choose between a single or multiprocessor system. This project uses the processor for simple data transfer and FPGA reconfiguration, therefore the single processor option is sufficient for use in the design. The *System Configuration* screen is shown in Figure B.8.

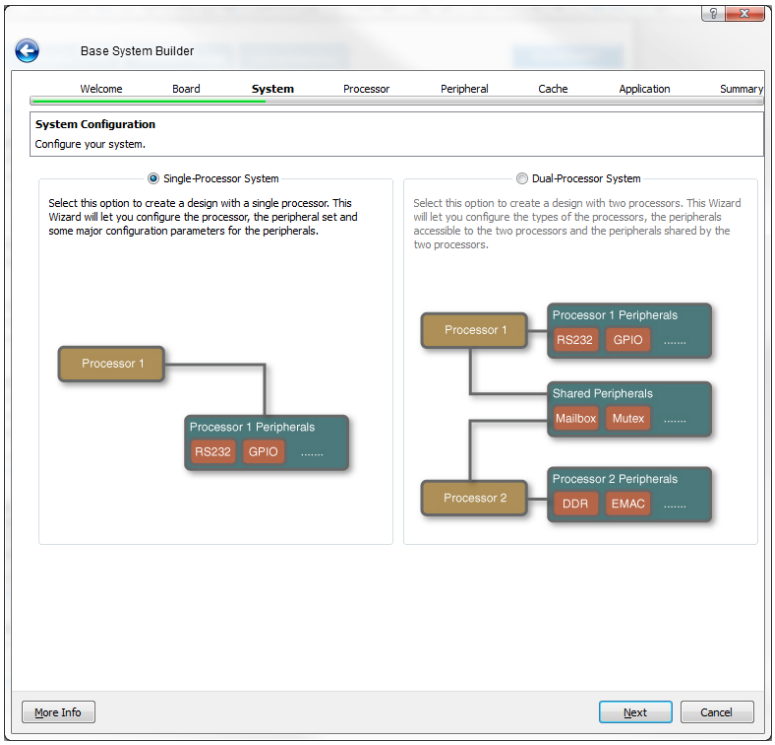

Figure B.8 XPS BSB System Configuration Window

The next configuration page is the *Processor Configuration* page. On this page the user can select which processor type to use (Microblaze or PowerPC), the system clock frequency, and local memory size. This project uses a Microblaze processor type running at 100 MHz. 100 MHz was chosen for the reference clock frequency to match the maximum operating frequency of the internal reconfiguration unit, HWICAP. The processor configuration page is shown Figure B.9.

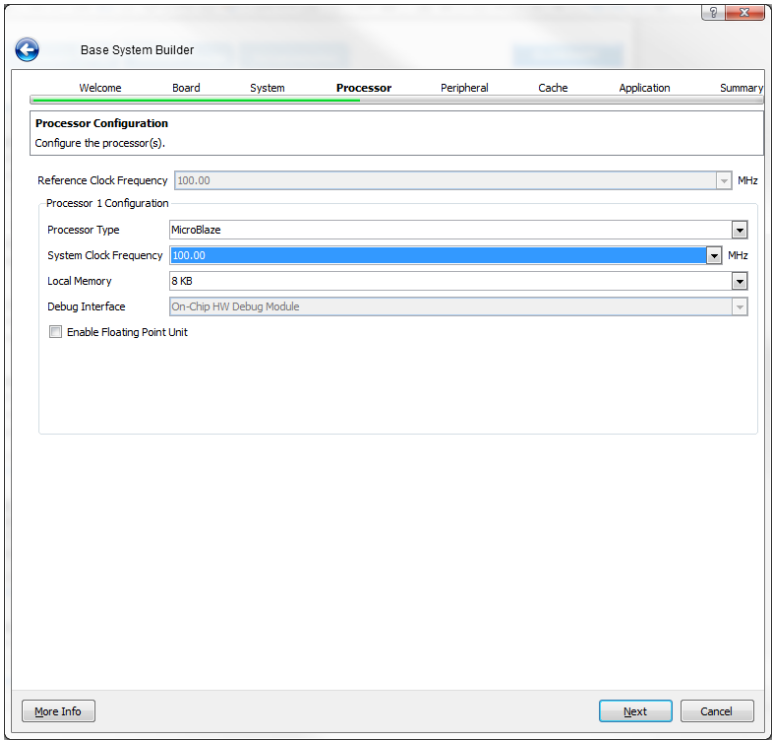

Figure B.9 XPS BSB Processor Configuration Window

After configuring the processor properties, the user is presented with a page to configure the peripherals that will be automatically attached to the system. The list of included peripherals will be based on the repository included in the previous step. The following automatically included peripherals were unnecessary for this application and therefore removed from the system: DIP_Switches_8Bit, Hard_Ethernet_MAC, IIC_EEPROM, LEDs_8Bit, LEDs_Positions, PCIe_Bridge, Push_Buttons_5Bit, and SysACE_CompactFlash. The RS232 ports were maintained for debugging purposes and were configured as xps_uart16550. Furthermore, an external system timer, xps_timer, was added to aid in timing analysis. The *Peripheral Configuration* window is shown in Figure B.10.

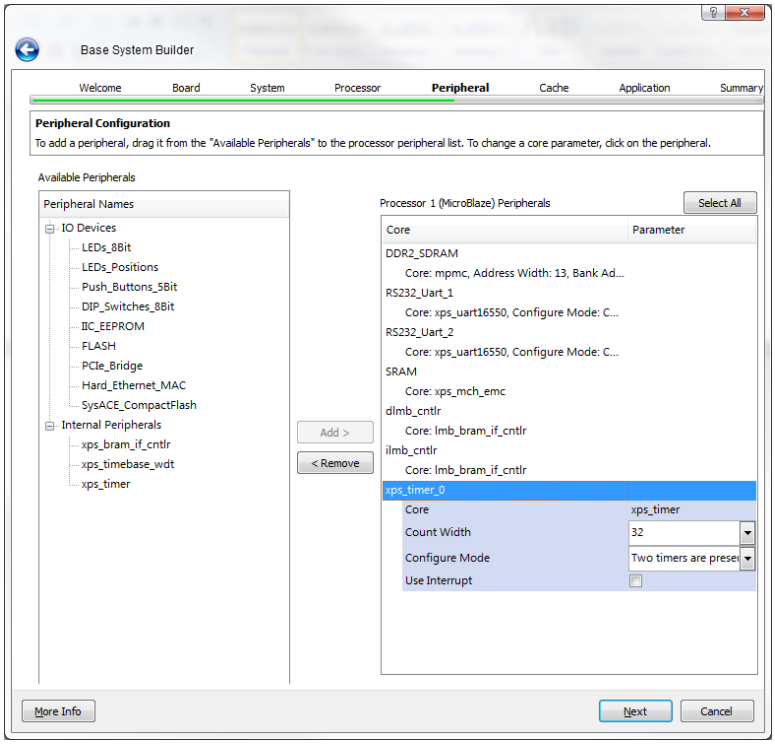

Figure B.10 XPS BSB Peripheral Configuration Window

The next two configuration pages allow the user to define custom caches and test applications respectively. These options were left as default. The final configuration page is a summary of the configured system. Selecting “Finish” will save the system settings into a .bsb (Base System Builder) file that can be recovered and reused on future projects.

B.3.2 Additional Peripheral Insertion (HWICAP)

The next step in setting up the system for partial reconfiguration is to add the reconfiguration peripheral to the design. The peripheral is listed as the “FPGA Internal Configuration Access Port” and can be found in the IP Catalog under the FPGA Reconfiguration heading. After selecting the IP, right-click and select Add IP to insert the HWICAP into the system design. Before the peripheral is added, a window will appear

that allows the user to configure the optional settings associated with it. In the case of the HWICAP, the only option modified is the write FIFO depth. This setting is increased to the maximum of 1024 for timing considerations that will be addressed in the timing analysis section of this document. The HWICAP configuration window is shown in Figure B.11.

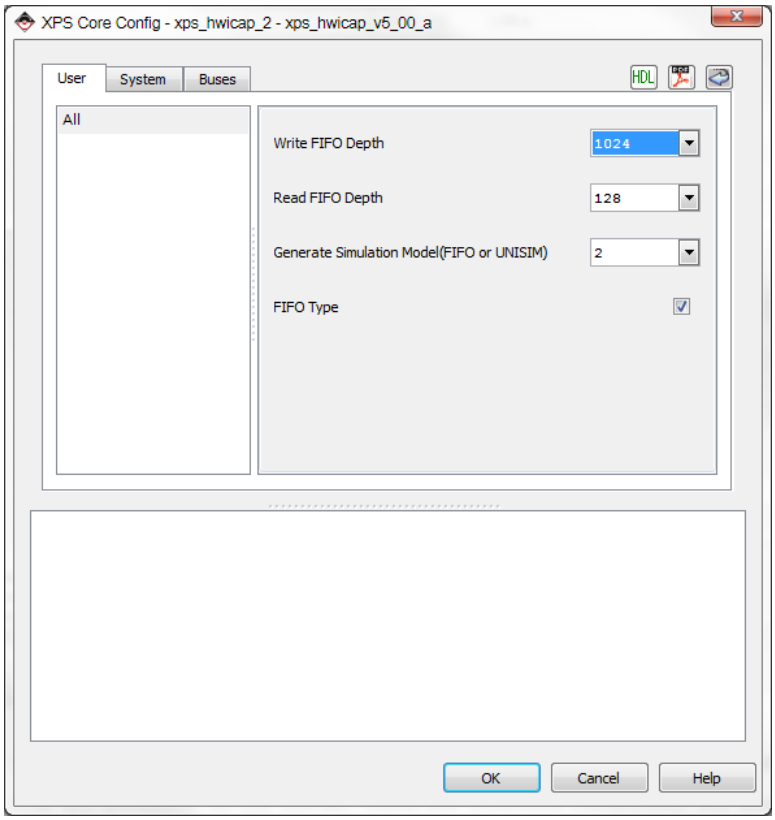

Figure B.11 XPS HWICAP Core Configuration Window

After the IP is added to the design, it must be manually connected to the other modules in the system. In the “Bus Interfaces” tab of the System Assembly View, expand *xps_hwicap_0* and in the drop-down menu for the SPLB select mb_plb to connect the

HWICAP to the PLB bus. Next, in the Ports tab of the System Assembly View, expand *xps_hwicap_0* and select the 100 MHz clock (clk_100_0000MHzPLL0) for the ICAP_Clk port. This application uses the polled version of the ICAP, therefore, the IP2INTC_Irpt port is left unconnected. Finally, the IP must be given a system memory address. In the Addresses tab of the System Assembly View, click the "Generate Addresses" button and verify that the xps_hwicap_0 listing is moved from the Unmapped Addresses section to the Microblaze's Address Map.

B.3.3 Create a Custom Peripheral

The final step in setting up the system is creating and adding a custom peripheral to the design. This custom peripheral will allow the user to include custom logic that can communicate with the other modules in the system. As described in the System Architecture section above, this allows the application (CODEC) to communicate to the Microblaze that a new RM is needed and for the Microblaze to signal the application that an RM is ready for use.

To create a custom peripheral, select Hardware -> Create or Import Peripheral… and click "Next". In the Peripheral Flow window, select "Create templates for a new peripheral" and advance to the next window. In the *Repository or Project* window, the user can select whether the IP should be saved to the local project or to an external repository for reuse in other applications. This peripheral is only intended to be used once and is therefore saved to the local project folder. The next window lets the user name and assign revision information to the peripheral. Revision information is extremely important for custom logic IPs because every time the peripheral is modified, it must be

re-imported to the EDK system design using a new revision number. Figure B.12 shows the *Name and Version* window for the first revision of the Decoder peripheral.

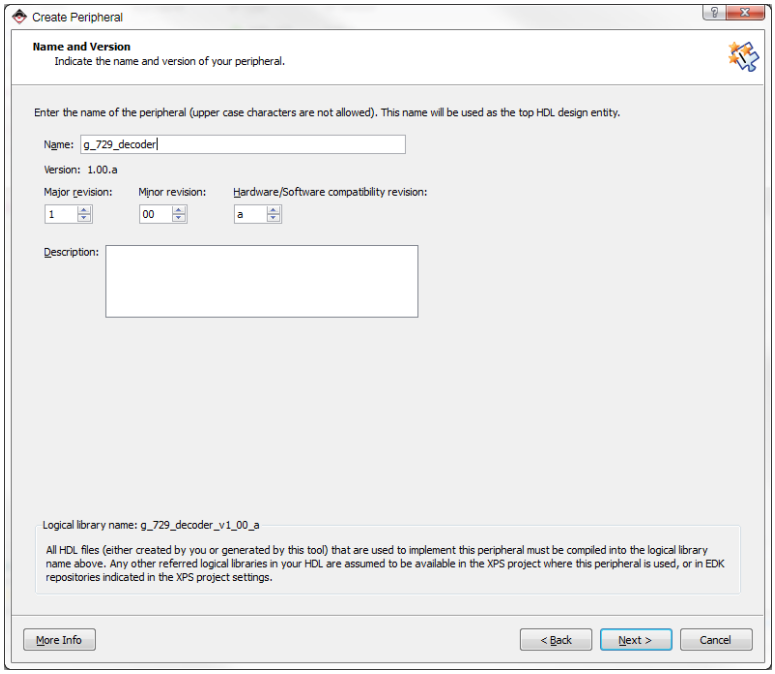

Figure B.12 XPS Create Peripheral *Name and Version* Window

The next window, Bus Interface, allows the user to select the bus type the IP will connect to. This system uses the PLB bus so it is selected. The following window allows the user to customize the IPs bus interface. The default selections include software registers and a data phase timer. However, the decoder implements its own memory. Therefore, the "User logic memory space" option must also be selected. The *IPIF Services* window is shown in Figure B.13.

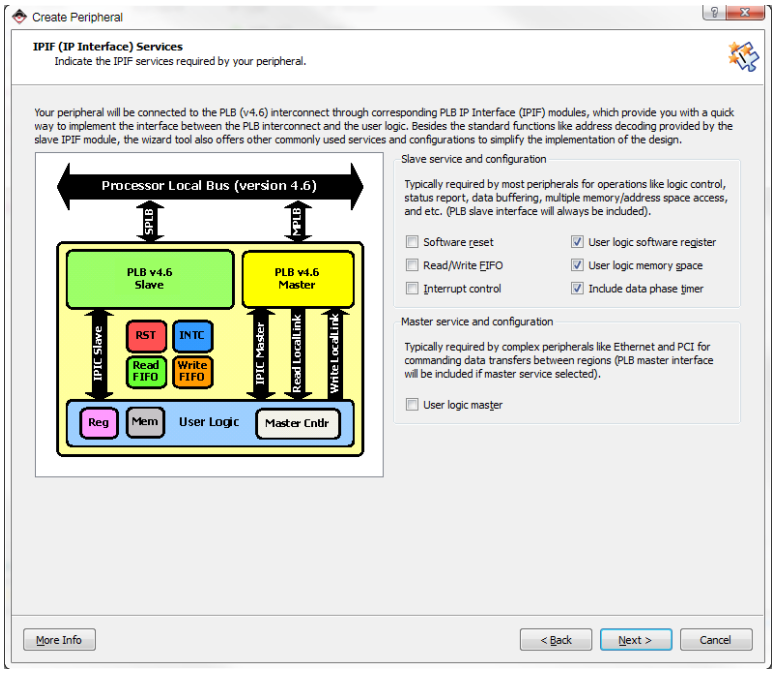

Figure B.13 XPS Create Peripheral *IPIF Services* Window

After configuring the IP interface, the user has the option to create and modify a slave interface for the peripheral. This functionality is unnecessary for this application and is left to default values. The next window, *User S/W Register*, permits the user to define the number of software accessible registers. These registers will be used to communicate the outputs from the decoder to the Microblaze. For this application, 32 registers are declared to encompass all outputs from the decoder and allow for future expansion if necessary. Then, in the *User Memory Space*, the user can define the number of memory regions needed for the peripheral. The decoder only requires one memory block. The *User S/W Register* and *User Memory Space* windows are shown in Figure B.14 and Figure B.15, respectively.

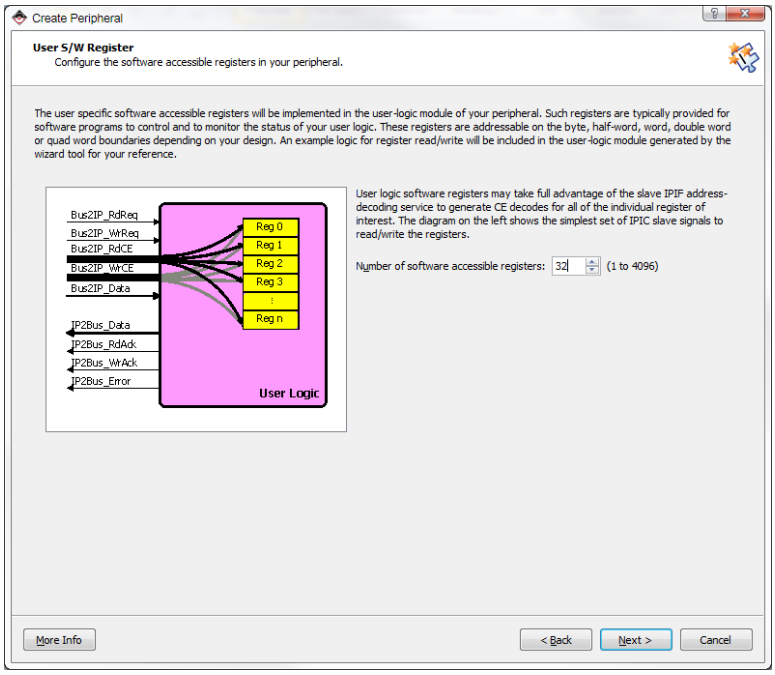

Figure B.14 XPS Create Peripheral *User S/W Register* Window

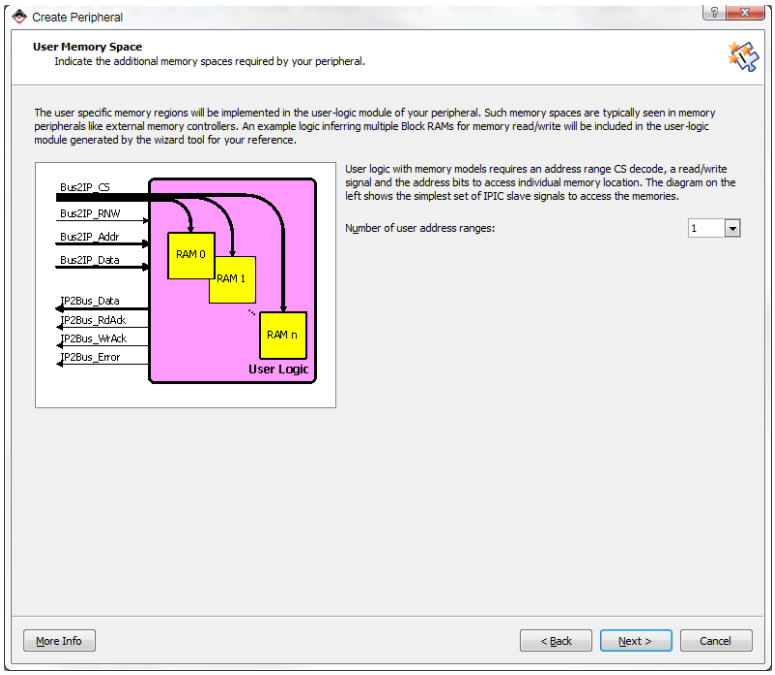

Figure B.15 XPS Create Peripheral *User Memory Space* Window

The next two windows, *IP Interconnect* and *Peripheral Simulation Support*, are left to default values. The final configuration window, *Peripheral Implementation Support*, allows the user to customize the automatically generated IP code. This project is written in Verilog, therefore, the "Generate stub 'user_logic' template in Verilog instead VHDL" is selected to make implementation simpler. Moreover, the "Generate ISE and XST project files to help you implement the peripheral using XST flow" option is selected to simplify the IP modification and verification process. The *Peripheral Implementation Support* window is shown in Figure B.16.

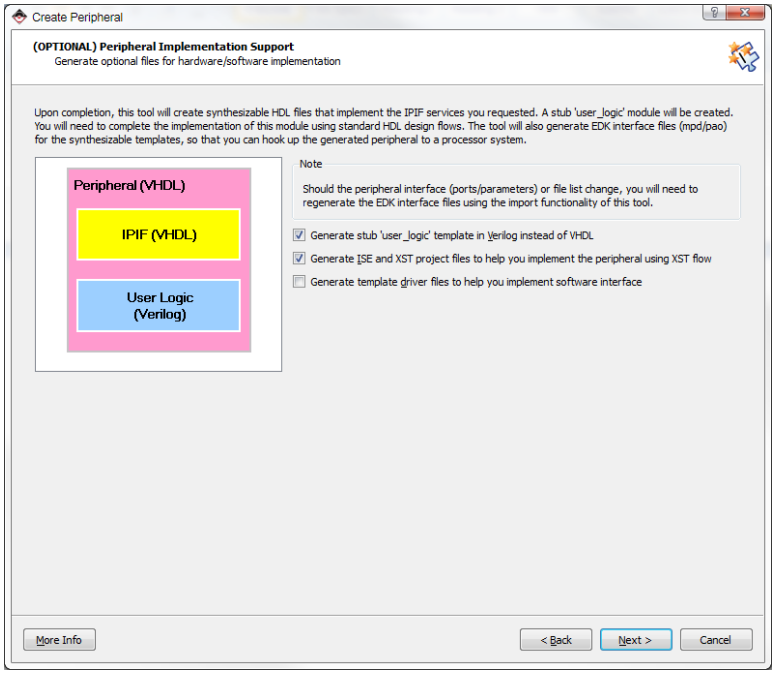

Figure B.16 XPS Create Peripheral Peripheral Implementation Support Window

Clicking "Finish" in the summary window will generate the IP and the project files associated with it. Furthermore, the peripheral will be added to the IP catalog under

the Project Local PCores -> USER group. Following the procedures outlined for adding the HWICAP peripheral in the previous step, the new custom IP can be added to the design, however, this is unnecessary at this time because the peripheral will be modified and re-imported under a different version name.

B.4 ISE - Custom Peripheral Modification

The next process in setting up the DPR system is to modify the custom logic IP template to include the Decoder code, as well as, update the interface code between the Decoder and the PLB bus. This process is completed in a new instance of the Xilinx ISE Project Navigator. The automatically generated project file is located in the \pcores\g_729_decoder_v1_00_a\devl\projnav\ directory of the EDK Project Root Folder.

Once the project is open, the decoder source files can be added to the project using ISE's source import tools. The top_level module must be instantiated in the user_logic.v source file for the logic to be included in the peripheral. Further modifications to the user_logic.v template include: decoder output signal definitions, decoder memory access controller instantiation, and connections from the software registers to the read/write bus signals. Once the peripheral is completed, the project is synthesized. Note that functional validation of the application being inserted into the peripheral template must be done in a separate external project. However, any modification to the application will require the peripheral project to be re-synthesized with the new application source files.

B.5 XPS – System Design II

Because the custom logic peripheral has been modified, it must be re-imported into the system design. This process is completed using the previous instance of the EDK in Xilinx XPS. To begin the re-import process, select Hardware -> Create or Import Peripheral... and click “Next”. In the Peripheral Flow window, select “Import existing peripheral” and click “Next”, shown in Figure B.17.

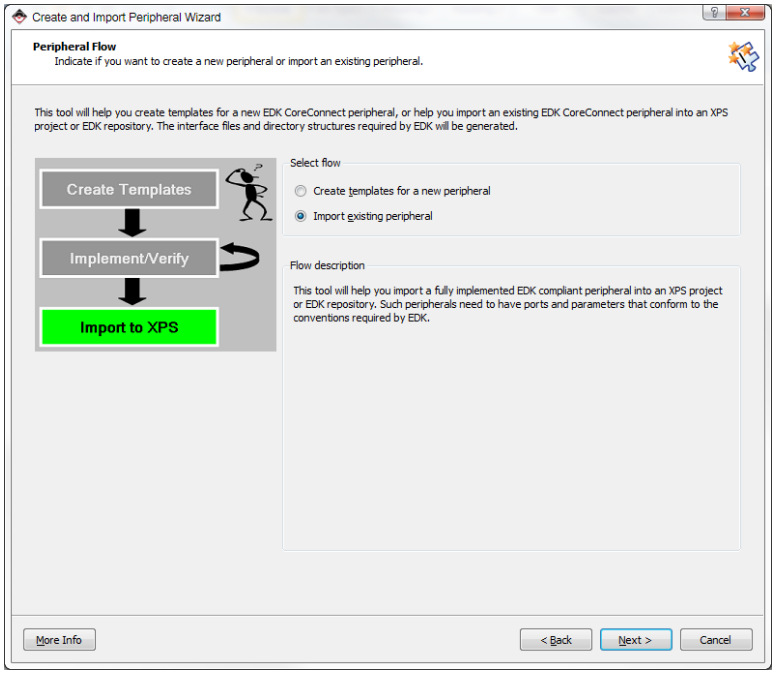

Figure B.17 XPS Import Peripheral *Peripheral Flow* Window

As in the creation process, this peripheral will be saved to the local project; therefore, the default value on the *Repository or Project* window is left selected. In the *Name and Version* window, the previously created peripheral name is selected from the drop-down menu and a new version number is assigned as shown in Figure B.18.

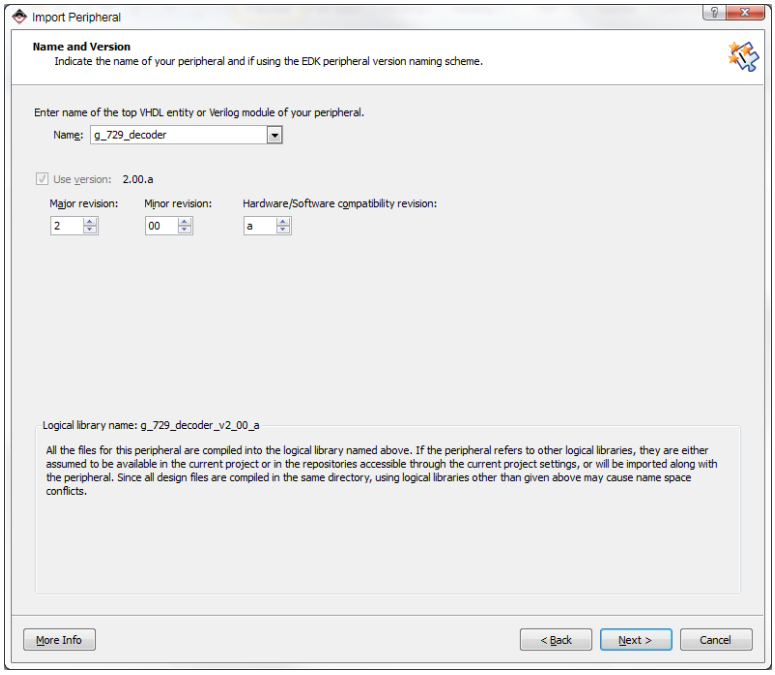

Figure B.18 XPS Import Peripheral *Name and Version* Window

The next window, *Source File Types*, lets the user select the types of source files that will be included in the peripheral. In this project, because memory cores are used in the peripheral, both “HDL source files” and “Netlist files” are selected. The *Source File Types* window is shown in Figure B.19.

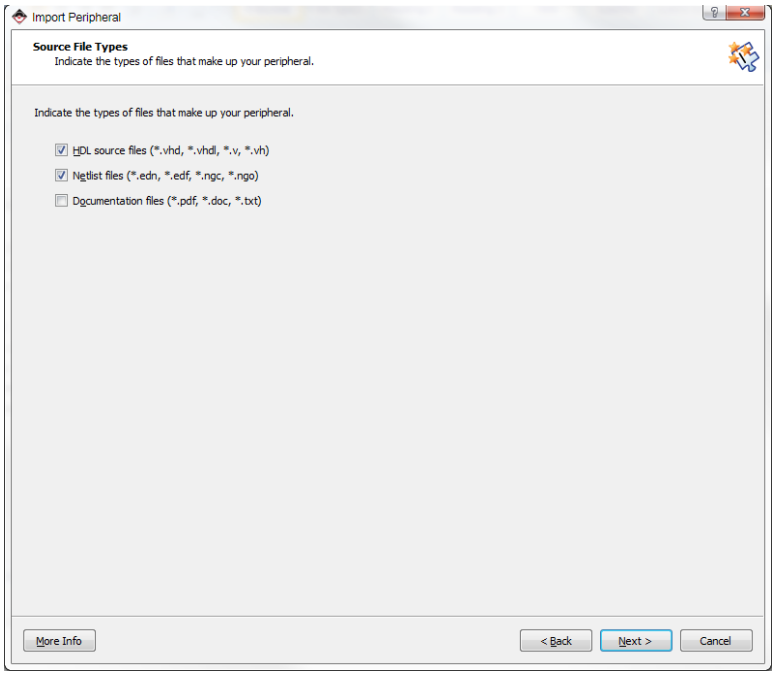

Figure B.19 XPS Import Peripheral *Source File Types* Window

The *HDL Source Files* window allows the user to select where the HDL source files will be scraped from. The HDL Language selection is changed to "Mixed" because this project uses both Verilog and VHDL source files. The most stable way to scrape the source files is by using the project file associated with the peripheral. This file is found in the \pcores\g_729_decoder_v1_00_a\devl\projnav\ directory of the EDK Project Root Folder. Note that there is another .prj file in the ..\devl\synthesis\ folder. This project file does not have the information necessary to import the peripheral correctly and should not be used. The *HDL Source Files* window is shown in Figure B.20.

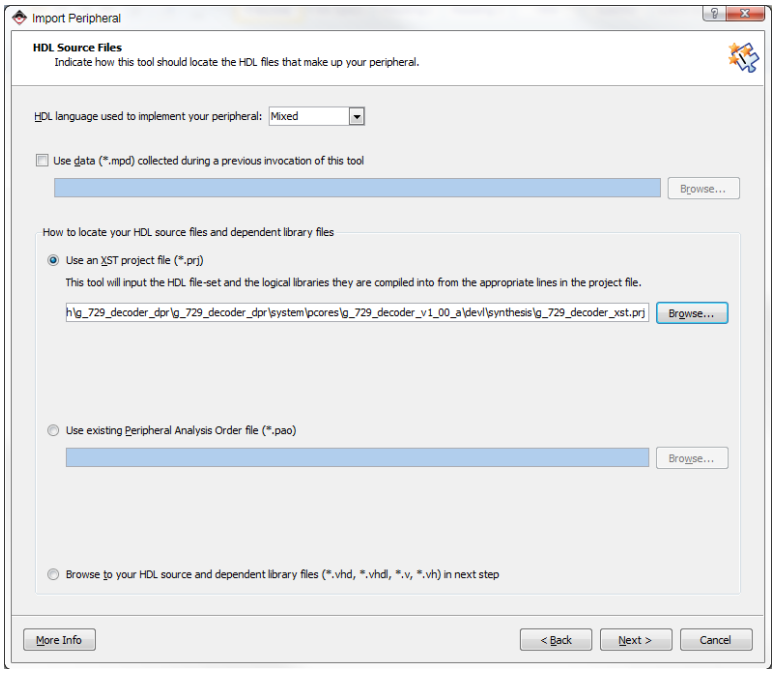

Figure B.20 XPS Import Peripheral *HDL Source Files* Window

In the subsequent *HDL Analysis Information* window, the results of the file scrape can be verified to ensure that all of the new source files have been identified.

The following window is the *Bus Interfaces* window that lets the user define the bus interface used by the peripheral. In this application, the peripheral is defined to be a "PLBV46 Slave (SPLB)" as illustrated in Figure B.21.

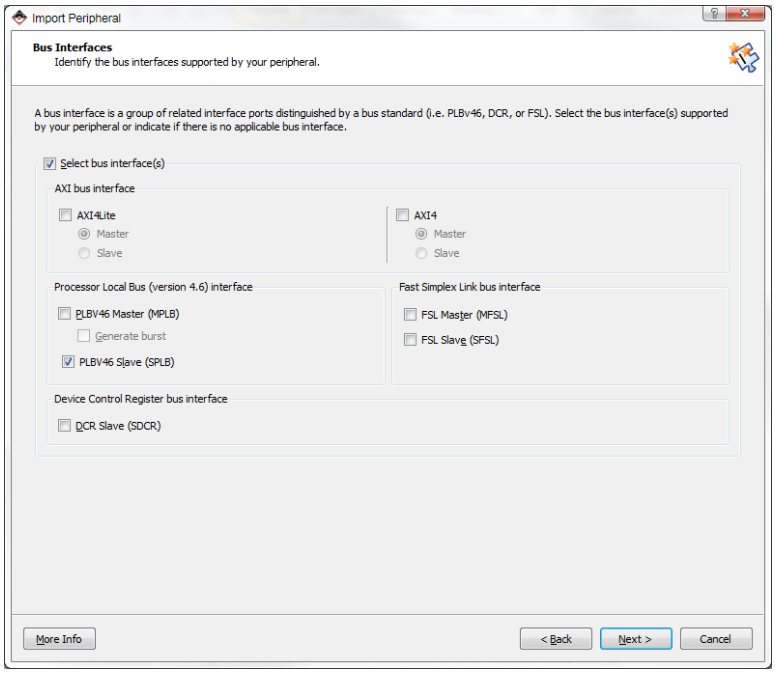

Figure B.21 XPS Import Peripheral *Bus Interfaces* Window

The subsequent *SPLB: Port* window summarizes the bus connections found by the tool. No custom bus connections were created for this application, so no modifications are required in this window. However, the *SPLB: Parameter* window does require modification. In the “Memory Space” pane a listing must be created for the memory core defined by the peripheral. Clicking the “Add” button will add a template listing to the pane. In the “Base Address Parameter” column select “C_MEM0_BASEADDR” from the drop-down menu. Likewise, in the “High Address Parameter” column select “C_MEM0_HIGHADDR”. This window is shown in Figure B.22.

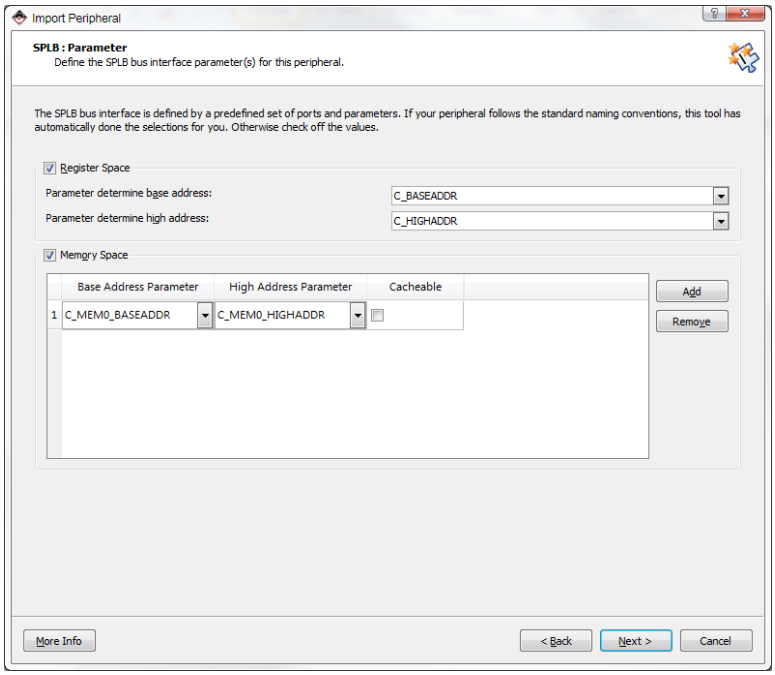

Figure B.22 XPS Import Peripheral *SPLB: Parameter* Window

The next window allows the user to configure interrupts for the peripheral. This peripheral does not use interrupts, therefore the "Select and configure interrupt(s)" option is deselected. The next two windows, *Parameter Attributes* and *Port Attributes* are unneeded and left as default.

The final window is the *Netlist Files* window that lets users include netlist files to the peripheral definition. This window is where the memory core netlist files are included. The netlist files are located in the \pcores\g_729_decoder_v1_00_a\hdl\verilog\ directory of the EDK Project Root Folder. The *Netlist Files* window is shown in Figure B.23.

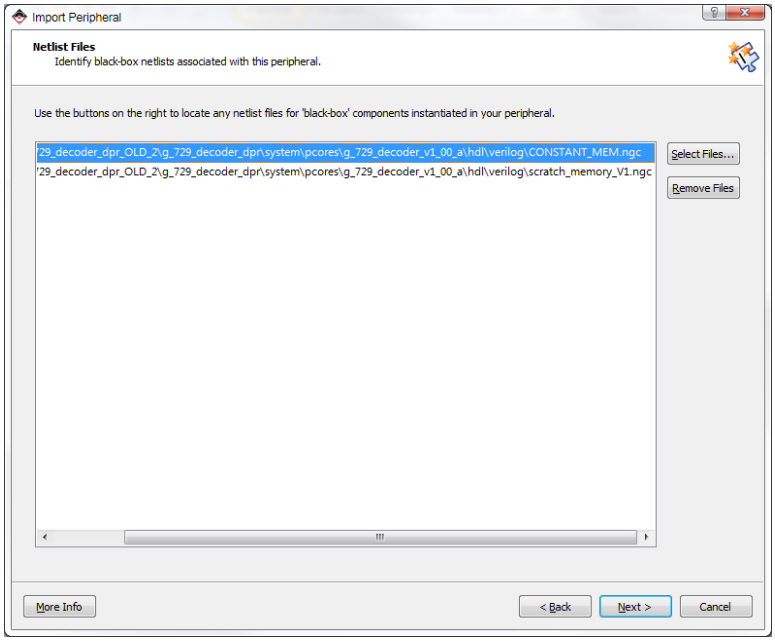

Figure B.23 XPS Import Peripheral *Netlist Files* Window

Clicking “Finish” on the summary window will generate a new peripheral based on the updated information that will be available in the IP catalog under Project Local PCores -> USER. The custom logic IP can now be added to the system design using the “add peripheral” process described in section B.3.2. When generating addresses for the custom IP, two different listings will be present, one for the IP and one for the memory space defined for it. The memory space listing does not have a default size and will produce an error during address generation. A size of 64k is defined for the memory space and regeneration of the address map will remove any generation errors.

B.6 ISE - Project Synthesis

Upon completing the design of the system using the EDK, the project must be synthesized for use in PlanAhead. This process is completed by the original instance of

Xilinx ISE used in section B.2. The decoder uses shared verilog parameter files. Therefore, before the project can be synthesized, the parameter files must be copied from the original peripheral folder to the new peripheral version's hdl folder. Failure to do this will result in an NgdBuild Error with an error code of 76. Once the files are copied, the synthesis process can be initiated by selecting the system in the Implementation pane and clicking "Synthesize - XST" in the Design tab.

B.7 PlanAhead - Floorplanning and Bitfile Generation

The final process in developing a DPR system is to configure the PR regions, implement the design, and generate the full and partial bitstreams. This process is completed using Xilinx PlanAhead 12.4 and follows closely to the processes found in [19]. Note: In order to develop and generate partial bitfiles, the *Partial Reconfiguration License* must be acquired from Xilinx. This license was obtained on a trial basis for Academic research purposes.

B.7.1 Create a Project

The first step in the PlanAhead process is to create a new project. Open PlanAhead and select "Create New Project". The program will open a wizard to configure the project. Click "Next", enter a project name and directory to save the project in, and click "Next" to advance. In the *Design Source* window, the "Specify synthesized (EDIF or NGC) netlist" option is selected, however, the "Set PR Project" option must also be selected to allow for partial reconfiguration options to become available (this option would be blocked if the PR License was not available). The *Design Source* window is shown in Figure B.24.

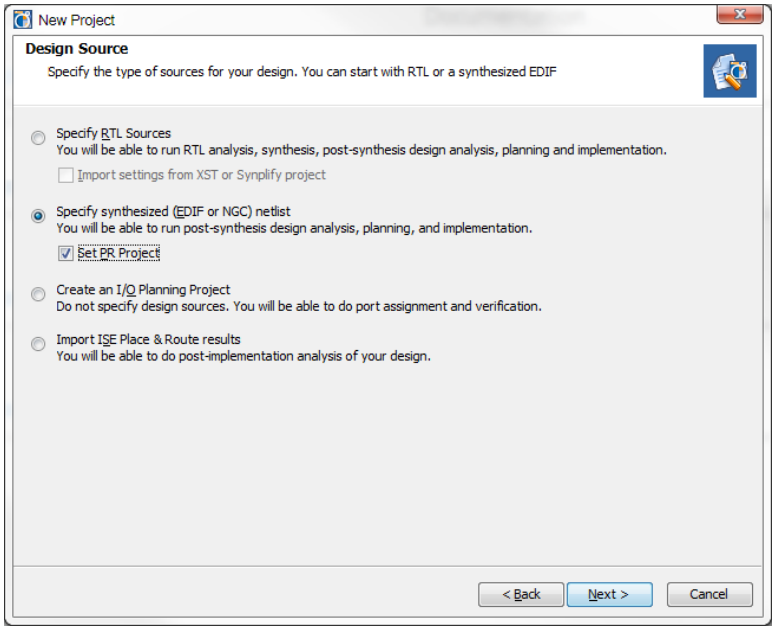

Figure B.24 PlanAhead New Project *Design Source* Window

The next window is the *Specify Top Netlist File* window, shown in Figure B.25, which allows users to identify the netlist file that defines the system in development. The netlist file to be selected is the system.ngc file created by the synthesis process completed in the previous section. It can be found in the ISE Project Root Folder.

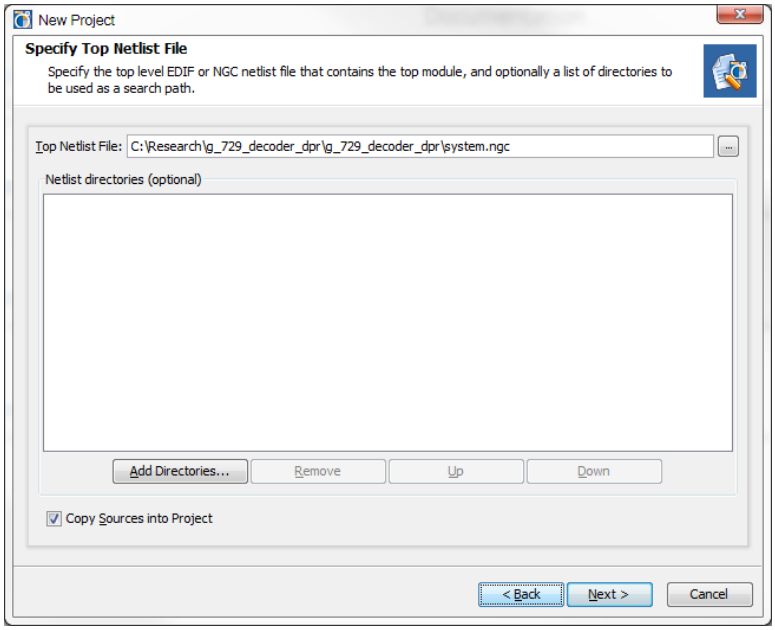

Figure B.25 PlanAhead New Project *Specify Top Netlist File* Window

The subsequent window allows the user to include constraints files that may be associated with the system included in the previous window. The EDK automatically creates a constraints file for the system and this system.ucf file is included here. It is found in the "data" folder of the EDK Project Root Folder. Also, a constraints file for the DDR2 SDRAM IP is provided by Xilinx and must be included in the project to prevent future errors. The DDR2 constrains file can be found in the "implementation" folder of the EDK Project Root Folder. The system.ucf file is marked as the target UCF to indicate it should be loaded with the netlist file. The *Add/Create Constraints* window is shown in Figure B.26.

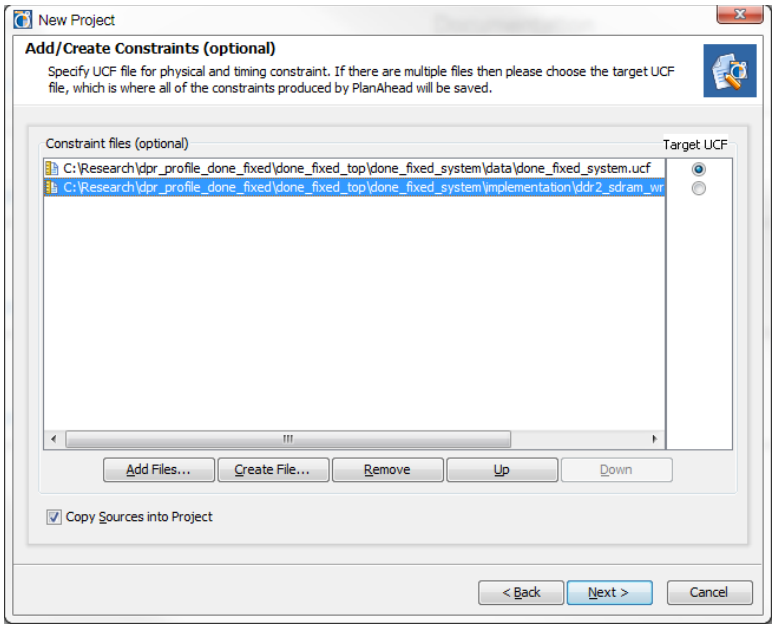

Figure B.26 PlanAhead New Project *Add/Create Constraints* Window

The next window, *Default Part*, requires the user to select the hardware that the project is targeting. This selection is used for floorplanning and size/usage data as well as implementation processes. The part information is extracted from the netlist file and does not require modification. The final window is a summary window. Clicking "Finish" in the summary window will import the netlist and constraints files and load the project.

B.7.2 Define a Reconfigurable Partition

The second step of the PlanAhead DPR process is to create and configure the reconfigurable partition of the design. To begin, click the "Netlist Design" button to load the netlist. This process will produce a warning, shown in Figure B.27, which indicates an undefined module was found and converted to a black box. This undefined module is the reconfigurable module defined inside the decoder and the warning is expected.

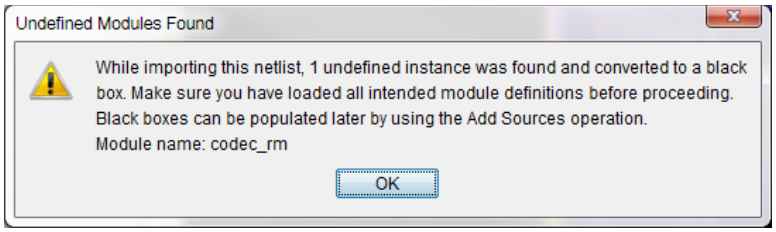

Figure B.27 PlanAhead Undefined Instance Warning

With the netlist loaded, select the PR module named i_codec_rm under g_729_decoder_0 in the Netlist tab as shown in Figure B.28.

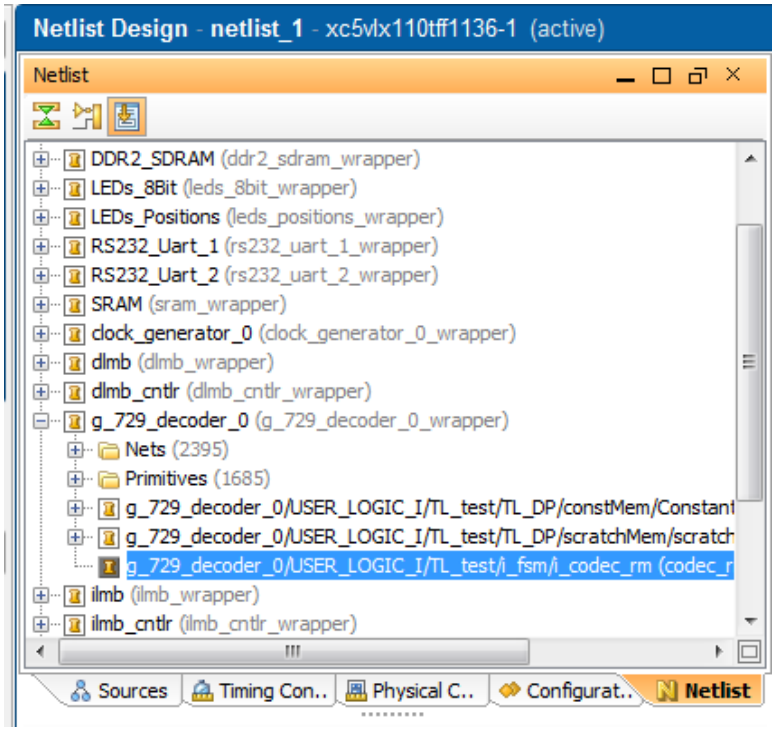

Figure B.28 PlanAhead Undefined Module in Netlist Tree

Right click on the module and select “Set Partition” to open a configuration window. In the first window, the module will be shown to be recognized as a reconfigurable partition. Click “Next” to continue.

The *Reconfigurable Module Name* window allows the user to name the RM currently being defined for this partition. The option indicating this module has an existing netlist is selected because this is an actual module. If a black box module were being added to allow for periods were no logic is implemented in the region, the second option would be selected. An example is shown in Figure B.29 where the first reconfigurable module, b1, is being added.

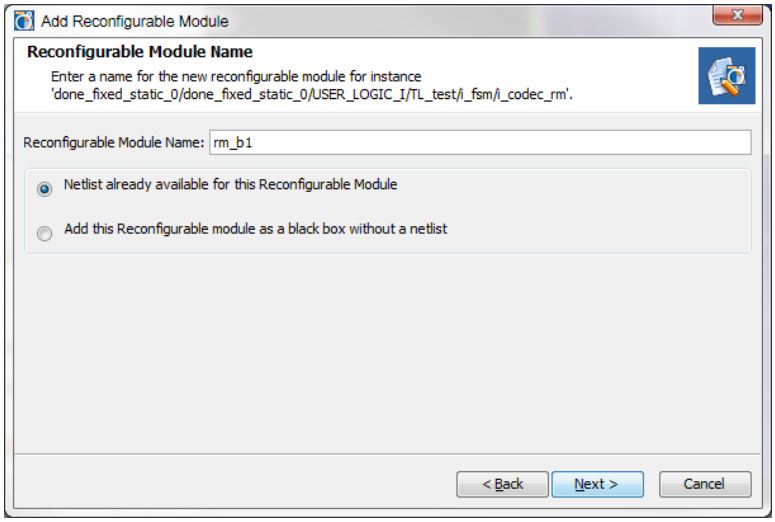

Figure B.29 PlanAhead Add RM - *RM Module Name* Window

The subsequent window allows the user to specify the netlist defining the reconfigurable module. In this project, the netlists for the RMs are found in the “ngc

files" folder in the Unit Module Synthesis Root Folder. This window is shown in Figure B.30.

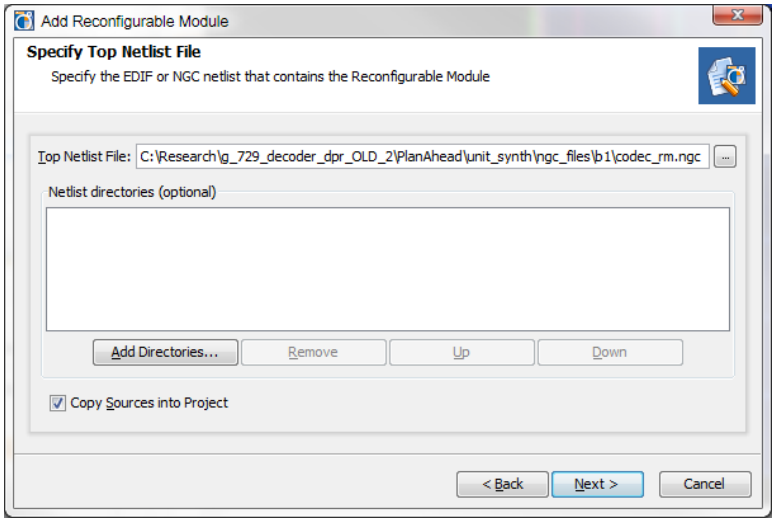

Figure B.30 PlanAhead Add RM – *Specify Top Netlist File* Window

The next window allows the user to add constraints files to associate with the module. The modules used in this application do not require constraints files, therefore, this window will be left to default. The final window summarizes the configuration process and clicking the "Finish" button creates the reconfigurable partition and adds the RM to it. The process can be verified by checking that the icon used to represent the module changes from a gold capital I on a black background to a gold diamond on a white background. The next step is to add the other reconfigurable modules by right clicking on the module and selecting "Add Reconfigurable Module…" then following the steps above.

B.7.3 Floorplanning

After defining the reconfigurable partition and adding the RMs to it, the physical region of the chip associated with the partition must be established. To accomplish this, select the partition under the Physical Constraints tab. Right click on the partition and select “Set Pblock Rectangle”. In the Device tab, draw a rectangle that encompasses the necessary resources to support the super-set of resources required by all of the reconfigurable modules, as shown in Figure B.31. To aid in this process, PlanAhead provides an estimation of the amount of resources required compared to the amount of resources selected by the current Pblock in the Pblock Properties pane under the Statistics tab, as shown in Figure B.32.

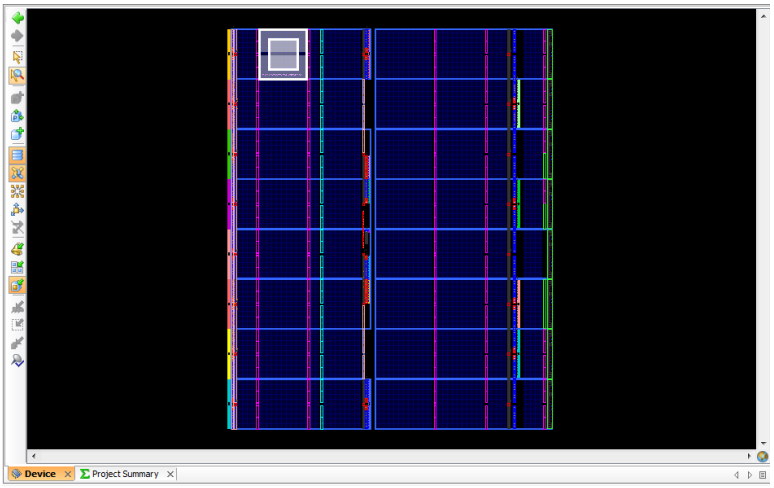

Figure B.31 PlanAhead Reconfigurable Partition Pblock

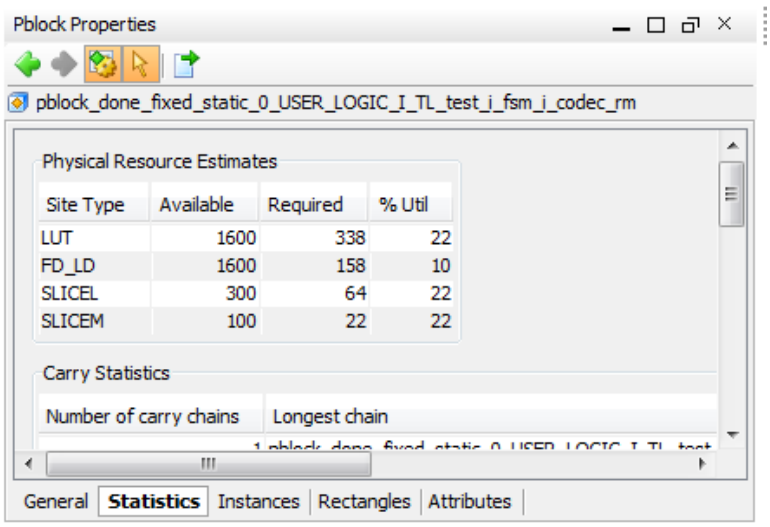

Figure B.32 PlanAhead Pblock Statistics Panel

It is of note that the size of the Pblock defined for this application is larger than required by the reconfigurable modules it serves. There are two factors for this design decision. The first is to aid in design implementation. The resource estimates give an estimate of logic resources required for a module but do not include the extra CLBs necessary for interconnections and partition interfaces. The second factor is bitfile size. The minimum reconfigurable region is 20 CLBs tall. Therefore, there is no reconfiguration speed benefit to reducing the rectangle below this height.

B.7.4 Design Implementation

After defining the partitions region, the design is ready to be implemented. The first step is to configure the first design run. In the “Design Runs” tab, select the first run. The run’s properties can be modified in the “Implementation Run Properties” panel. In the “Options” tab, the effort levels of the map and PAR procedures can be defined. In the “Partitions” tab, both the static logic and the RP must be set to “Implement”. When the

run's properties are set, the run can be initiated by right clicking on the run and selecting "Launch Runs…"

Once the first run is complete, PlanAhead will provide the user with a list of options for post run processes. Because other RMs must be implemented, "Promote Partitions" is selected. Then, to create additional runs, select "Create Multiple Runs" from the drop-down menu on the "Implement" button. This action will launch a wizard that will allow the user to add as many extra runs as necessary to accommodate the number of RMs in the design. The main window is the *Choose Implementation Strategies and Reconfigurable Modules*. In this window, runs can be added using the "More" button, renamed under the Name column, and modified by selecting the "…" button under the Partition Action column, as shown in Figure B.33. For each addition run created, the partition action required is to "Import" the Static Logic from the first run and "Implement" the new RM. This configuration is illustrated in Figure B.34.

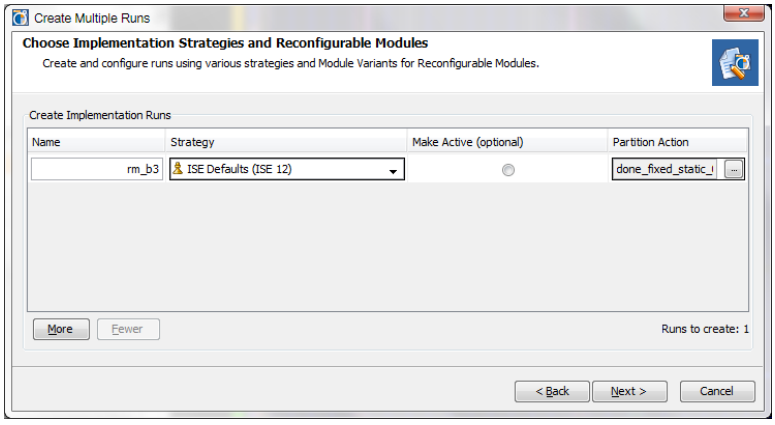

Figure B.33 PlanAhead Create Multiple Runs – Choose Implementation Strategies and RMs Window

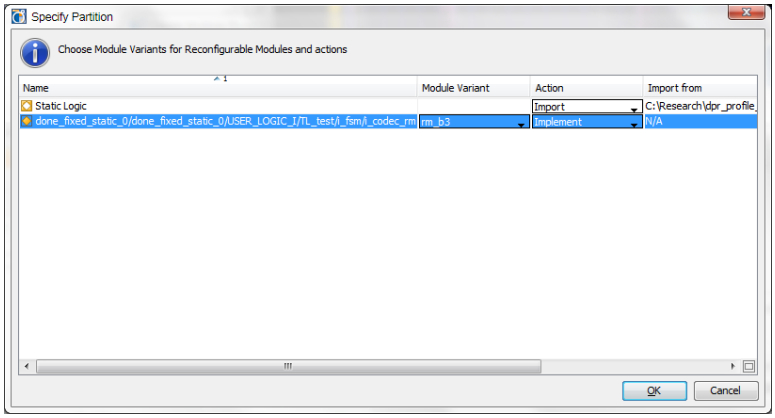

Figure B.34 PlanAhead Create Multiple Runs – *Specify Partition* Window

In the final window of the wizard, the newly created runs can be launched.

B.7.5 Generate Bitstreams

The final step of the PlanAhead procedure is to generate the bitstreams for each run. To perform this task, select all of the implemented runs and right click to select "Generate Bitstream". This will launch the bitgen process for each run and will, upon completion, create two bitstreams for each run. The first bitstream is a full bitstream that defines the entire FPGA with the run's RM implemented in the RP. The second bitfile is a partial bitfile that defines the run's RM for partial reconfiguration into the RP during execution. These bitfiles will be saved in the Project.runs folder in the PlanAhead Project Folder. For ease of access, the bitfiles are relocated to the Project.bit folder.

B.8 XPS – Test Platform Initialization

Now that the bitfiles have been generated, the test application to run the system can be developed. This test application is created and run in Xilinx's SDK. To launch the SDK with the system's properties automatically imported, the instance of Xilinx XPS

used in section B.3and B.5 is used. In XPS, select Project -> Export Hardware Design to SDK… In the opened window, select “Export and Launch SDK”. This will open an instance of the SDK.

B.9 SDK – Test Platform Configuration

The Xilinx SDK allows the user to design, implement, and execute applications to run on the Microblaze implemented on the FPGA. The Xilinx SDK version used in this project is 12.4. In order to develop and run an application in the SDK, the user must first create a new project, set up the board support package (BSP), and configure the console output.

To create a new project, select File -> New -> Xilinx C Project. This will open a wizard. The first window will allow the user to name the project and select a project template to use. For this project, the “Hello World” template was selected because it automatically establishes file connections with minimal code to modify. Furthermore, it can be run “as-is” to test that the following configuration steps are completed successfully. The first window is shown in Figure B.35.

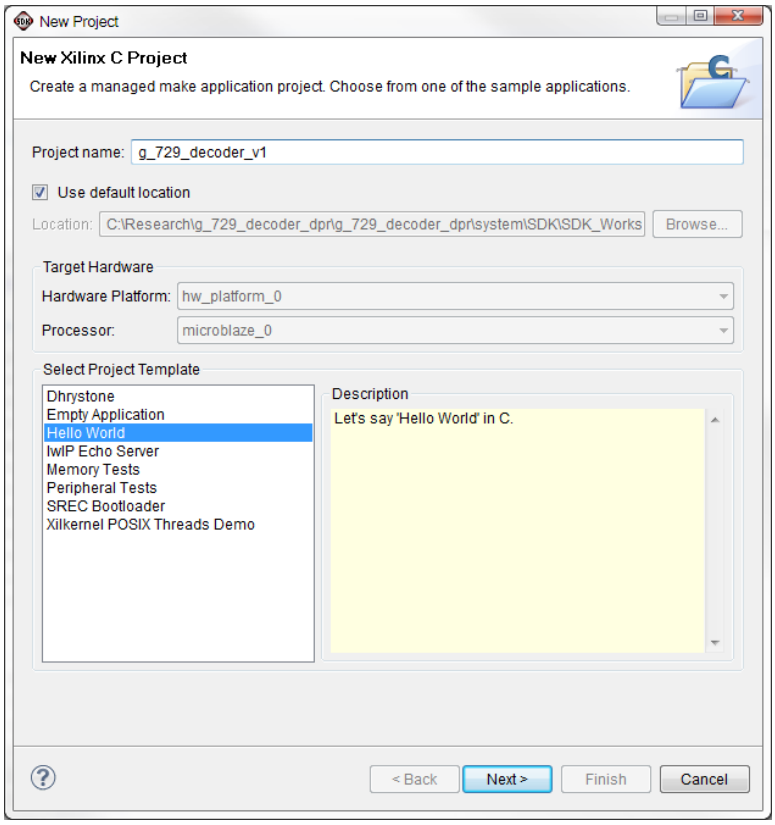

Figure B.35 SDK New Project – Project Name and Template Window

The next window in the project creation wizard is the BSP configuration page. It allows you to name a new BSP and set its directory or load an existing BSP. For this project, a new BSP is created as shown in Figure B.36.

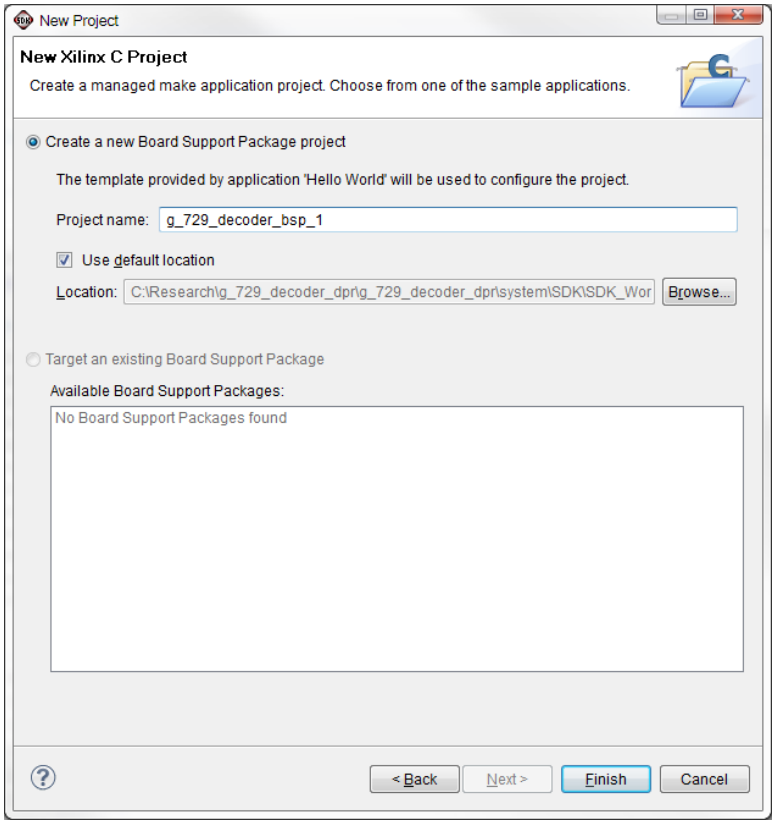

Figure B.36 SDK New Project – BSP Configuration Window

Clicking “Finish” will create the project and BSP as well as any associated files. At this point, the main source file will be named helloworld.c. This is refactored to mimic the project name by right clicking on the file in the Project Explorer panel and selecting Refactor.

The next step to setting up the SDK environment is to configure the BSP. Select Xilinx Tools -> Board Support Package Settings and click “OK” to select the project’s BSP. In the standalone tab, set the stdin and stdout values to mdm_0 to enable standard communication over the JTAG. This configuration is shown in Figure B.37.

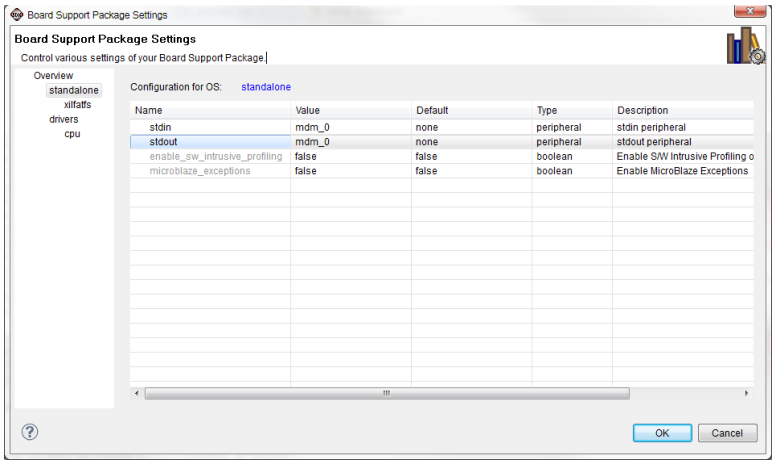

Figure B.37 SDK BSP Configuration Window

The final step to setting up the SDK is connecting the SDK console to the JTAG so that the output from the Microblaze can be viewed in the SDK. To carry out this task, you must access the run configurations; however, there are no run configurations available until the program is executed. Therefore, click the run button (Green Triangle) and when you receive a query about configuring the FPGA as seen in Figure B.38, click cancel. This will allow you to now access the run configurations through Run -> Run Configurations.

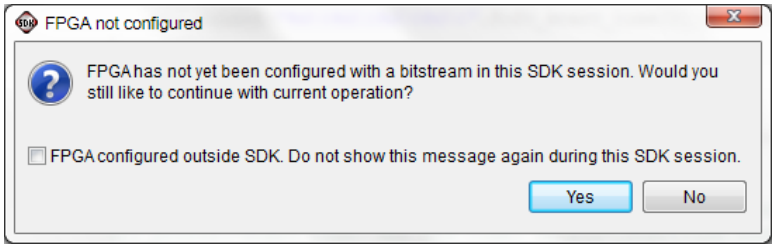

Figure B.38 SDK FPGA Not Configured Warning

In the Run Configurations menu, select the STDIO Connection tab and select the "Connect STDIO to Console" option. Finally, select the JTAG UART from the "Port" drop-down menu and click "Apply". This configuration is shown in Figure B.39.

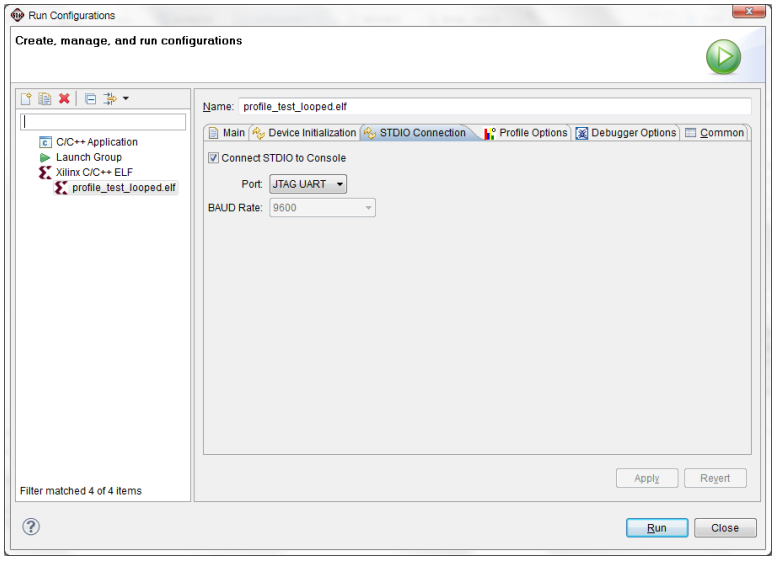

Figure B.39 SDK Run Configurations - STDIO Connection Window

B.10 iMPACT – FPGA Programming

The final procedure in setting up the FPGA for a DPR application is programming the board. This procedure is completed using the Xilinx iMPACT tool. As the program is launched it will provide the user with a list of processes to execute. Because the board will be manually programmed, the "Configure devices using Boundary-Scan (JTAG)" option is selected as shown in Figure B.40.

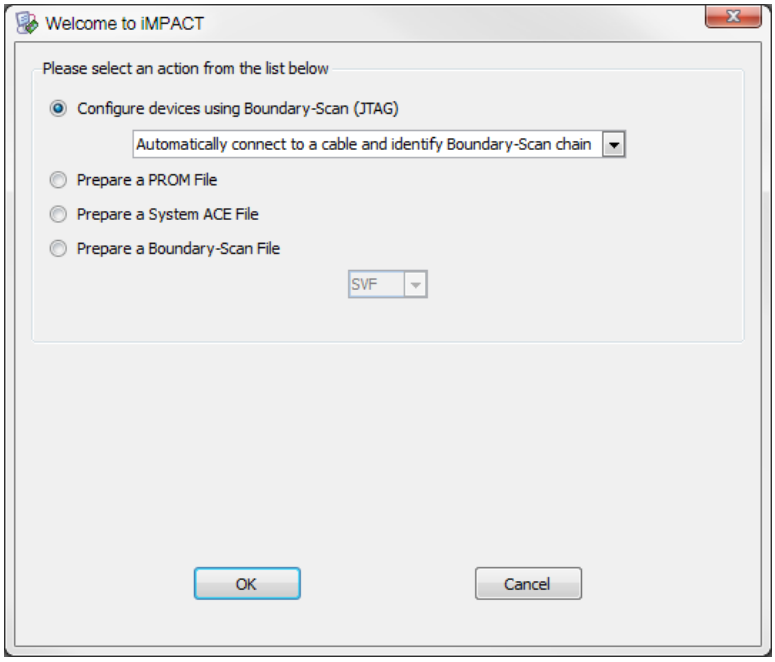

Figure B.40 iMPACT Launch Window

After the program automatically detects the device chain, it will query the user to assign configuration files. For this project only the FPGA needs to be programmed; therefore, “No” is selected. This will load the main window shown in Figure B.41.

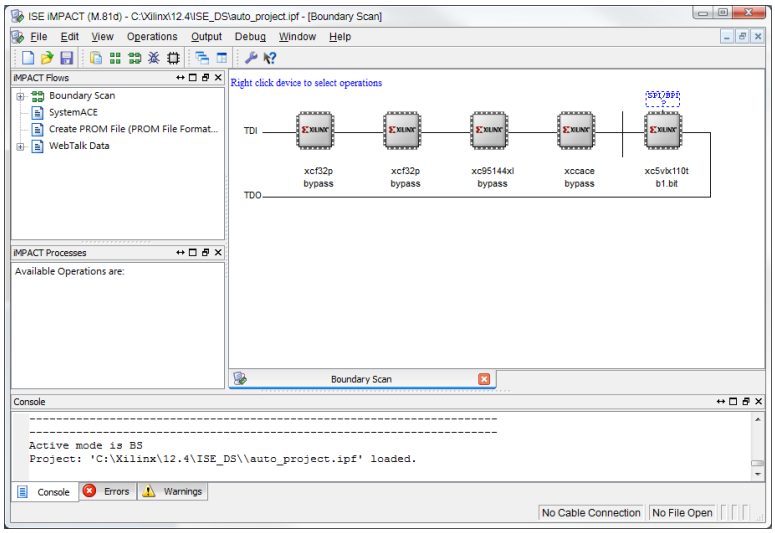

Figure B.41 iMPACT Device Chain

After the device chain is loaded, the configuration file (bitfile) can be associated with the FPGA. To do this, right click on the FPGA icon (xc5vlx110t) and select “Assign New Configuration File…” iMPACT will open a file browser that allows the user to select the bitfile for programming. The bitfiles are located in the Project.bit folder in the PlanAhead Project Folder. Once the configuration file is loaded, the FPGA can be programmed by right clicking on the FPGA icon and selecting “Program”.

This completes the software flow for developing a DPR system and setting up a test platform for it. Further discussion on the modification of the decoder application as well as development of the test application is given in CHAPTER II and CHAPTER III, respectively.

APPENDIX C

DYNAMIC PARTIAL RECONFIGURATION DECODER SYSTEM TESTING APPLICATION

```
[1] /*
[2] profile_test.c: simple test application
[3] */

[4] #include <stdio.h>
[5] #include <stdlib.h>
[6] #include "platform.h"
[7] #include "xparameters.h"
[8] #include "xutil.h"
[9] #include "xuartns550.h"
[10]    #include "xuartns550_l.h"
[11]    #include "xbasic_types.h"
[12]    #include "xhwicap.h"
[13]    #include "xhwicap_i.h"
[14]    #include <xstatus.h>
[15]    #include "platform_config.h"
[16]    #include "string.h"
[17]    #include "xio.h"
[18]    #include "xil_io.h"
[19]    #include "xtmrctr.h"
[20]    #include "bitfile1.h"
[21]    #include "bitfile2.h"
[22]    #include "serial_data.h"
```

```
[23]  #include "timer_vars.h"

[24]  /* Constant Definitions */
[25]  #define num_run                          14
[26]  #define XPAR_DEFAULT_BAUD_RATE   115200
[27]  #define STATIC_BASE_ADDR
          XPAR_DONE_FIXED_STATIC_0_BASEADDR
[28]  #define STATIC_MEM_BASE_ADDR
          XPAR_DONE_FIXED_STATIC_0_MEM0_BASEADDR

[29]  /* Function Declarations */
[30]  int load_rm(Xuint32 *rm_load_reg);
[31]  int decode_frame(Xuint32 *debug_32_reg, Xuint32 *start_reg, Xuint32
      *done_reg, Xuint32 *rm_load_reg, Xuint32 *rm_ready_reg, Xuint32 *state_reg,
      Xuint32 *test_cont, Xuint32 *mux_reg, Xuint32 *CPP_done_reg, Xuint32
      *DLSP_done_reg, Xuint32 *QLPC_done_reg, Xuint32 *LD8K_done_reg, Xuint32
      *RESIDU_done_reg, Xuint32 *WEIGHT_AZ_done_reg, Xuint32
      *PST_LTP_done_reg, Xuint32 *CALC_ST_FILT_done_reg, Xuint32
      *FILT_MU_done_reg, Xuint32 *SCALE_ST_done_reg, Xuint32
      *POST_PROCESS_done_reg, Xuint32 *SYN_FILT_done_reg, Xuint32
      *COPY_done_reg);
[32]  int load_bitfile(XHwIcap *HwIcap_Ptr, Xuint32 *bit_mem, Xuint32 word_pairs,
      Xuint32 odd_words);
```

```
[33]    void send_byte(XUartNs550 *uart, unsigned char c);
[34]    Xuint8 recv_byte(XUartNs550 *uart);
[35]    int DeviceWrite(XHwIcap *InstancePtr, u32 *FrameBuffer, u32 NumWords);

[36]    /* Global Variables */
[37]    struct rm_data_struct {
[38]    Xuint32 *mem_locs[NUM_RM];
[39]    Xuint32 word_pair_counts[NUM_RM];
[40]    Xuint8 odd_words[NUM_RM];
[41]    } rm_data;

[42]    Xuint32 rm = 0;

[43]    /* HwIcap instance */
[44]    static XHwIcap          HwIcap;

[45]    int main()
[46]    {
[47]    /*****************************************************************
    **
[48]    *
[49]    *
            [1] Platform Setup
```

```
[50]    *
[51]    *
[52]    ******************************************************************
        /
[53]    init_platform();

[54]    print("Hello World\n\r");

[55]    int Status;
[56]    int i,j,k;
[57]    i = 0;
[58]    j = 0;
[59]    k = 0;

[60]    /* Configure the Timer */
[61]    timer_0 = &xps_timer_0;
[62]    Status = XTmrCtr_Initialize(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
[63]    if (Status != XST_SUCCESS) {
[64]    return XST_FAILURE;
[65]    }
[66]    XTmrCtr_Start(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
[67]    StartTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
[68]    EndTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
```

```
[69]  Calibration = EndTime - StartTime;
[70]  print("Timer Initialized!\r\n");

[71]  /* Configure the HwIcap */
[72]  XHwIcap_Config *ConfigPtr;
[73]  ConfigPtr = XHwIcap_LookupConfig(XPAR_XPS_HWICAP_0_DEVICE_ID);
[74]  if (ConfigPtr == NULL) {
[75]  return XST_FAILURE;
[76]  }
[77]  Status = XHwIcap_CfgInitialize(&HwIcap, ConfigPtr,ConfigPtr->BaseAddress);
[78]  if (Status != XST_SUCCESS) {
[79]  return XST_FAILURE;
[80]  }
[81]  print("HwIcap Initialized\r\n");

[82]  Xuint32 num_rms;
[83]  num_rms = 0x00000002;

[84]  for(i=0;i<(int)num_files;i++)
[85]  {
[86]  Xuint8 rm_number;
[87]  if(i==0)
[88]  rm_number = 1;
```

```
[89]    else if(i==1)
[90]    rm_number = 3;

[91]    xil_printf("File: %d\r\n",rm_number);
[92]    rm_data.mem_locs[rm_number] = NULL;
[93]    rm_data.word_pair_counts[rm_number] = 7471;
[94]    rm_data.odd_words[rm_number] = 1;

[95]    rm_data.mem_locs[rm_number] = (Xuint32 *)malloc(80000* sizeof(int));
[96]    if(rm_data.mem_locs[rm_number]==NULL)
[97]    {
[98]    xil_printf("ALLOCATION ERROR! Could not allocate memory for file
    %d",rm_number);
[99]    return -1;
[100]   }
[101]   else
[102]   {
[103]   print("Memory Allocation Successful\r\n");
[104]   }

[105]   /* Transfer bitfile data */
[106]   for(j=0;j<(rm_data.word_pair_counts[rm_number]*2)+rm_data.odd_words[rm_n
    umber];j++)
```

```
[107]   {
[108]   rm_data.mem_locs[rm_number][j] = 0x00000000;

[109]   if(rm_number == 1)
[110]   rm_data.mem_locs[rm_number][j] = bitfile1_data[j];
[111]   else if(rm_number == 3)
[112]   rm_data.mem_locs[rm_number][j] = bitfile2_data[j];

[113]   if(j%1000==0) {
[114]   xil_printf("Read word %d\r\n",j);
[115]   }
[116]   }
[117]   }

[118]   /*************************************************************
        **
[119]   *
[120]   *
[121]   Main Program
[122]   *
[123]   *
[124]   *************************************************************
        /
```

```
[125]  /* Decoder External Signals and Memory Locations Declarations */
[126]  Xuint32 *start_reg, *done_reg, *rm_ready_reg, *rm_load_reg, *state_reg,
       *mux_reg, *debug_32_reg;
[127]  Xuint32 *CPP_done_reg, *DLSP_done_reg, *QLPC_done_reg,
       *LD8K_done_reg, *RESIDU_done_reg;
[128]  Xuint32 *WEIGHT_AZ_done_reg, *PST_LTP_done_reg,
       *CALC_ST_FILT_done_reg, *FILT_MU_done_reg;
[129]  Xuint32 *SCALE_ST_done_reg, *POST_PROCESS_done_reg,
       *SYN_FILT_done_reg, *COPY_done_reg;
[130]  Xuint32 *test_cont;
[131]  Xuint32 *mem, *serial, *parm, *aq_t_low, *aq_t_high, *synth_buf, *old_exc,
       *lsp_new;
[132]  Xuint32 *voicing, *t0_first, *mem_syn, *pst_out;
[133]  Xuint32 *gain_prec, *y2_hi, *y2_lo, *y1_hi, *y1_lo, *x0, *x1;

[134]  /* Decoder Memory Location Offset Definitions */
[135]  Xuint32 serial_offset = 2944;
[136]  Xuint32 parm_offset = 624;
[137]  Xuint32 aq_t_low_offset = 800;
[138]  Xuint32 aq_t_high_offset = 816;
[139]  Xuint32 synth_buf_offset = 1024;
[140]  Xuint32 old_exc_offset = 3072;
```

```
[141]  Xuint32 lsp_new_offset = 384;
[142]  Xuint32 voicing_offset = 3567;
[143]  Xuint32 t0_first_offset = 3564;
[144]  Xuint32 mem_syn_offset = 4080;
[145]  Xuint32 pst_out_offset = 1152;
[146]  Xuint32 gain_prec_offset = 1115;
[147]  Xuint32 y2_hi_offset = 1116;
[148]  Xuint32 y2_lo_offset = 1117;
[149]  Xuint32 y1_hi_offset = 1118;
[150]  Xuint32 y1_lo_offset = 1119;
[151]  Xuint32 x0_offset = 1120;
[152]  Xuint32 x1_offset = 1121;

[153]  /* Decoder External Signals and Memory Locations Address Definitions */
[154]  mem = (Xuint32 *) (STATIC_MEM_BASE_ADDR);
[155]  serial = mem + serial_offset;
[156]  parm = mem + parm_offset;
[157]  aq_t_low = mem + aq_t_low_offset;
[158]  aq_t_high = mem + aq_t_high_offset;
[159]  synth_buf = mem + synth_buf_offset;
[160]  old_exc = mem + old_exc_offset;
[161]  lsp_new = mem + lsp_new_offset;
[162]  voicing = mem + voicing_offset;
```

```
[163]  t0_first = mem + t0_first_offset;
[164]  mem_syn = mem + mem_syn_offset;
[165]  pst_out = mem + pst_out_offset;
[166]  gain_prec = mem + gain_prec_offset;
[167]  y2_hi = mem + y2_hi_offset;
[168]  y2_lo = mem + y2_lo_offset;
[169]  y1_hi = mem + y1_hi_offset;
[170]  y1_lo = mem + y1_lo_offset;
[171]  x0 = mem + x0_offset;
[172]  x1 = mem + x1_offset;

[173]  start_reg =          (Xuint32 *) (STATIC_BASE_ADDR);
[174]  done_reg =           (Xuint32 *) (STATIC_BASE_ADDR + 4*1);
[175]  rm_ready_reg =             (Xuint32 *) (STATIC_BASE_ADDR +
    4*2);
[176]  rm_load_reg =        (Xuint32 *) (STATIC_BASE_ADDR + 4*3);
[177]  test_cont =          (Xuint32 *) (STATIC_BASE_ADDR + 4*4);
[178]  CPP_done_reg =             (Xuint32 *) (STATIC_BASE_ADDR +
    4*5);
[179]  DLSP_done_reg =      (Xuint32 *) (STATIC_BASE_ADDR + 4*6);
[180]  QLPC_done_reg =      (Xuint32 *) (STATIC_BASE_ADDR + 4*7);
[181]  LD8K_done_reg =      (Xuint32 *) (STATIC_BASE_ADDR + 4*8);
[182]  RESIDU_done_reg =    (Xuint32 *) (STATIC_BASE_ADDR + 4*9);
```

```
[183]  WEIGHT_AZ_done_reg =     (Xuint32 *) (STATIC_BASE_ADDR + 4*10);
[184]  PST_LTP_done_reg =       (Xuint32 *) (STATIC_BASE_ADDR + 4*11);
[185]  CALC_ST_FILT_done_reg =  (Xuint32 *) (STATIC_BASE_ADDR + 4*12);
[186]  FILT_MU_done_reg =               (Xuint32 *) (STATIC_BASE_ADDR +
       4*13);
[187]  SCALE_ST_done_reg =              (Xuint32 *) (STATIC_BASE_ADDR +
       4*14);
[188]  POST_PROCESS_done_reg =          (Xuint32 *) (STATIC_BASE_ADDR +
       4*15);
[189]  SYN_FILT_done_reg =              (Xuint32 *) (STATIC_BASE_ADDR +
       4*16);
[190]  COPY_done_reg =          (Xuint32 *) (STATIC_BASE_ADDR + 4*17);
[191]  state_reg =              (Xuint32 *) (STATIC_BASE_ADDR + 4*18);
[192]  mux_reg =                (Xuint32 *) (STATIC_BASE_ADDR + 4*19);
[193]  debug_32_reg =                   (Xuint32 *) (STATIC_BASE_ADDR +
       4*20);

[194]  /* Decoder Initial Conditions Definitions */
[195]  *voicing = 0x0000003C;
[196]  *gain_prec = 0x00004000;
[197]  *y2_hi = 0x00000000;
[198]  *y2_lo = 0x00000000;
[199]  *y1_hi = 0x00000000;
```

```
[200]   *y1_lo = 0x00000000;
[201]   *x0 = 0x00000000;
[202]   *x1 = 0x00000000;

[203]   /* Decoder Execution Loop */
[204]   for(i=0;i<num_run;i++)
[205]   {
[206]   /* Transfer New Audio Frame Data */
[207]   for(k = 0; k<80; k++)
[208]   {
[209]   serial[k] = serial1_data[i*80+k];
[210]   }

[211]   /* Decoder Execution Preparation */
[212]   *start_reg = 0;
[213]   *test_cont = 1;
[214]   *rm_ready_reg = 0;
[215]   XTmrCtr_Stop(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
[216]   XTmrCtr_Reset(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
[217]   XTmrCtr_Start(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
[218]   StartTime = 0;
[219]   EndTime = 0;
```

```
[220]   /* Launch Decoder */
[221]   Status = decode_frame(debug_32_reg, start_reg, done_reg, rm_load_reg,
        rm_ready_reg, state_reg, test_cont, mux_reg, CPP_done_reg, DLSP_done_reg,
        QLPC_done_reg, LD8K_done_reg, RESIDU_done_reg, WEIGHT_AZ_done_reg,
        PST_LTP_done_reg, CALC_ST_FILT_done_reg, FILT_MU_done_reg,
        SCALE_ST_done_reg, POST_PROCESS_done_reg, SYN_FILT_done_reg,
        COPY_done_reg);

[222]   /* Output timing data */
[223]   xil_printf("%d\t%d\t%d\t",i,StartTime,BeginReconfig[0]);
[224]   for(k=0;k<fifo_counter;k++)
[225]   {
[226]   xil_printf("%d\t%d\t%d\t%d\t",fifo_start_time[0][k],fifo_end_time[0][k],config_s
        tart_time[0][k],config_end_time[0][k]);
[227]   }
[228]   xil_printf("%d\t%d\t",EndReconfig[0],BeginReconfig[1]);
[229]   for(k=0;k<fifo_counter;k++)
[230]   {
[231]   xil_printf("%d\t%d\t%d\t%d\t",fifo_start_time[1][k],fifo_end_time[1][k],config_s
        tart_time[1][k],config_end_time[1][k]);
[232]   }
[233]   xil_printf("%d\t%d\r\n",EndReconfig[1],EndTime);
[234]   }
```

```
[235]   /* Platform cleanup */
[236]   XTmrCtr_Stop(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
[237]   cleanup_platform();

[238]   return 0;
[239]   }

[240]   int decode_frame(Xuint32 *debug_32_reg, Xuint32 *start_reg, Xuint32
        *done_reg, Xuint32 *rm_load_reg, Xuint32 *rm_ready_reg, Xuint32 *state_reg,
        Xuint32 *test_cont, Xuint32 *mux_reg, Xuint32 *CPP_done_reg, Xuint32
        *DLSP_done_reg, Xuint32 *QLPC_done_reg, Xuint32 *LD8K_done_reg, Xuint32
        *RESIDU_done_reg, Xuint32 *WEIGHT_AZ_done_reg, Xuint32
        *PST_LTP_done_reg, Xuint32 *CALC_ST_FILT_done_reg, Xuint32
        *FILT_MU_done_reg, Xuint32 *SCALE_ST_done_reg, Xuint32
        *POST_PROCESS_done_reg, Xuint32 *SYN_FILT_done_reg, Xuint32
        *COPY_done_reg)
[241]   {
[242]   int status;
[243]   rm = 0;
[244]   *start_reg = 1;
[245]   *start_reg = 0;
```

```
[246]   StartTime = XTmrCtr_GetValue(&xps_timer_0,
        XPAR_XPS_TIMER_0_DEVICE_ID);
[247]   do
[248]   {
[249]   if(*rm_ready_reg != *rm_load_reg)
[250]   {
[251]   if(*rm_load_reg <= 3)
[252]   {
[253]   fifo_counter = 0;
[254]   BeginReconfig[rm] = XTmrCtr_GetValue(&xps_timer_0,
        XPAR_XPS_TIMER_0_DEVICE_ID);
[255]   status = load_bitfile(&HwIcap, rm_data.mem_locs[*rm_load_reg],
        rm_data.word_pair_counts[*rm_load_reg], rm_data.odd_words[*rm_load_reg]);
[256]   EndReconfig[rm] = XTmrCtr_GetValue(&xps_timer_0,
        XPAR_XPS_TIMER_0_DEVICE_ID);
[257]   rm++;
[258]   if(status != 0)
[259]   return -1;
[260]   else
[261]   *rm_ready_reg = *rm_load_reg;
[262]   }
[263]   else
[264]   {
```

```
[265]   *rm_ready_reg = *rm_load_reg;
[266]   }
[267]   }
[268]   }while(*done_reg != 1);
[269]   EndTime = XTmrCtr_GetValue(&xps_timer_0,
        XPAR_XPS_TIMER_0_DEVICE_ID);
[270]   return 0;
[271]   }


[272]   int load_bitfile(XHwIcap *HwIcap_Ptr, Xuint32 *bit_mem, Xuint32 word_pairs,
        Xuint32 odd_words)
[273]   {
[274]   XStatus Status;

[275]   Status = DeviceWrite(HwIcap_Ptr, bit_mem,
        rm_data.word_pair_counts[1]*2+rm_data.odd_words[1]);
[276]   if (Status != XST_SUCCESS)
[277]   {
[278]   print("Error writing to ICAP!");
[279]   return -1;
[280]   }
```

```
[281]   return 0;
[282]   }

[283]   /* Xilinx Provided Function */
[284]   int DeviceWrite(XHwIcap *InstancePtr, u32 *FrameBuffer, u32 NumWords)
[285]   {

[286]   u32 WrFifoVacancy;
[287]   u32 IntrStatus;

[288]   Xil_AssertNonvoid(InstancePtr != NULL);
[289]   Xil_AssertNonvoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
[290]   Xil_AssertNonvoid(FrameBuffer != NULL);
[291]   Xil_AssertNonvoid(NumWords > 0);

[292]   /*
[293]   Make sure that the last Read/Write by the driver is complete.
[294]   */
[295]   if (XHwIcap_IsTransferDone(InstancePtr) == FALSE) {
[296]   return XST_FAILURE;
[297]   }

[298]   /*
```

[299] Check if the ICAP device is Busy with the last Read/Write

[300] */

[301] if (XHwIcap_IsDeviceBusy(InstancePtr) == TRUE) {

[302] return XST_FAILURE;

[303] }

[304] /*

[305] Set the flag, which will be cleared when the transfer

[306] is entirely done from the FIFO to the ICAP.

[307] */

[308] InstancePtr->IsTransferInProgress = TRUE;

[309] /*

[310] Disable the Global Interrupt.

[311] */

[312] XHwIcap_IntrGlobalDisable(InstancePtr);

[313] /*

[314] Set up the buffer pointer and the words to be transferred.

[315] */

[316] InstancePtr->SendBufferPtr = FrameBuffer;

```
[317]   InstancePtr->RequestedWords = NumWords;
[318]   InstancePtr->RemainingWords = NumWords;


[319]   /*
[320]   Fill the FIFO with as many words as it will take (or as many as we
[321]   have to send).
[322]   */
[323]   fifo_start_time[rm][fifo_counter] = XTmrCtr_GetValue(&xps_timer_0, 0);
[324]   WrFifoVacancy = XHwIcap_GetWrFifoVacancy(InstancePtr);
[325]   while ((WrFifoVacancy != 0) &&
[326]   (InstancePtr->RemainingWords > 0)) {

[327]   XHwIcap_FifoWrite(InstancePtr, *InstancePtr->SendBufferPtr);
[328]   InstancePtr->RemainingWords--;
[329]   WrFifoVacancy--;
[330]   InstancePtr->SendBufferPtr++;
[331]   }
[332]   fifo_end_time[rm][fifo_counter] = XTmrCtr_GetValue(&xps_timer_0, 0);

[333]   /*
[334]   Start the transfer of the data from the FIFO to the ICAP device.
[335]   */
```

[336] config_start_time[rm][fifo_counter] = XTmrCtr_GetValue(&xps_timer_0, 0);

[337] XHwIcap_StartConfig(InstancePtr);

[338] while ((XHwIcap_ReadReg(InstancePtr->HwIcapConfig.BaseAddress,

[339] XHI_CR_OFFSET)) &

[340] XHI_CR_WRITE_MASK);

[341] config_end_time[rm][fifo_counter] = XTmrCtr_GetValue(&xps_timer_0, 0);

[342] fifo_counter = 1;

[343] /*

[344] Check if there is more data to be written to the ICAP

[345] */

[346] if (InstancePtr->RemainingWords != NULL){

[347] /*

[348] Check whether it is polled or interrupt mode of operation.

[349] */

[350] if (InstancePtr->IsPolled == FALSE) { /* Interrupt Mode */

[351] /*

[352] If it is interrupt mode of operation then the

[353] transfer of the remaining data will be done in the

[354] interrupt handler.

[355] */

[356] /*

[357] Clear the interrupt status of the earlier interrupts

[358] */

[359] IntrStatus = XHwIcap_IntrGetStatus(InstancePtr);

[360] XHwIcap_IntrClear(InstancePtr, IntrStatus);

[361] /*

[362] Enable the interrupts by enabling the

[363] Global Interrupt.

[364] */

[365] XHwIcap_IntrGlobalEnable(InstancePtr);

[366] }

[367] else { /* Polled Mode */

[368] while (InstancePtr->RemainingWords > 0) {

[369] fifo_start_time[rm][fifo_counter] = XTmrCtr_GetValue(&xps_timer_0, 0);

[370] WrFifoVacancy =

[371] XHwIcap_GetWrFifoVacancy(InstancePtr);

[372] while ((WrFifoVacancy != 0) &&

```
[373]   (InstancePtr->RemainingWords > 0)) {
[374]   XHwIcap_FifoWrite(InstancePtr,
[375]   *InstancePtr->SendBufferPtr);

[376]   InstancePtr->RemainingWords--;
[377]   WrFifoVacancy--;
[378]   InstancePtr->SendBufferPtr++;
[379]   }
[380]   fifo_end_time[rm][fifo_counter] = XTmrCtr_GetValue(&xps_timer_0, 0);

[381]   config_start_time[rm][fifo_counter] = XTmrCtr_GetValue(&xps_timer_0, 0);
[382]   XHwIcap_StartConfig(InstancePtr);
[383]   while ((XHwIcap_ReadReg(
[384]   InstancePtr->HwIcapConfig.BaseAddress,
[385]   XHI_CR_OFFSET)) & XHI_CR_WRITE_MASK);
[386]   config_end_time[rm][fifo_counter] = XTmrCtr_GetValue(&xps_timer_0, 0);

[387]   fifo_counter++;

[388]   }

[389]   /*
[390]   Clear the flag to indicate the write has been done
```

```
[391]   */
[392]   InstancePtr->IsTransferInProgress = FALSE;
[393]   InstancePtr->RequestedWords = 0x0;
[394]   }

[395]   } else {

[396]   /*
[397]   Clear the flag to indicate the write has been done
[398]   */
[399]   InstancePtr->IsTransferInProgress = FALSE;
[400]   InstancePtr->RequestedWords = 0x0;
[401]   }

[402]   return XST_SUCCESS;
[403]   }
```