BitMaT - Bitstream Manipulation Tool for Xilinx FPGAs

Casey J Morford

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Dr. Cameron Patterson, Chair
Dr. Thomas Martin
Dr. Michael Hsiao

December 15th, 2005
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: FPGA, Xilinx, Virtex-II, Virtex-II Pro, Dynamic Module Server, Bitstream Manipulation, Bitstream, Partial Reconfiguration

BitMaT - Bitstream Manipulation Tool for Xilinx FPGAs

Casey J. Morford

(ABSTRACT)

With the introduction of partially reconfigurable FPGAs, we are now able to perform dynamic changes to hardware running on an FPGA without halting the operation of the design. Module based partial reconfiguration allows the hardware designer to create multiple hardware modules that perform different tasks and swap them in and out of designated dynamic regions on an FPGA. However, the current mainstream partial reconfiguration flow provides a limited and inefficient approach that requires a strict set of guidelines to be met. This thesis introduces BitMaT, a tool that provides the low-level bitstream manipulation as a member tool of an alternative, automated, modular partial reconfiguration flow.

Dedication

I'd like to dedicate this thesis to my wife, Beth, for her love, support and encouragement throughout this journey. To my six month old son, Miles, who is one of the most wonderful joys in my life. Also to my parents, Laura Morford and John Morford for their never ending love and support of everything I do.

Acknowledgments

I would like to thank Dr. Cameron Patterson, my academic advisor, for his support during my time at Virginia Tech, and for giving me the opportunity to work on a project that resulted in this thesis. His guidance and encouragement made this work possible.

My sincere thanks also goes to Dr. Peter Athanas who's knowledge and background helped guide me through this research and for his determination to white-board every wall of the Configurable Computing Laboratory.

In addition to my committee chair, I would like to thank Dr. Tom Martin and Dr. Thomas Hsiao for serving on my committee and providing me with some of my more interesting and beneficial courses in my college career.

Thanks to Justin Rice and Todd Fleming whom also worked on the Dynamic Module Server project. I wish you both luck with the pursuit of your PhDs.

Thanks to Jonathan Graf at Luna Innovations for your support, encouragement and leadership on the Dynamic Module Server project.

I would also like to thank the members of the Configurable Computing and E-Textiles Labs whom I've become friends with. Good luck to all of you and your future endeavors.

This project was funded by the AFRL under a sub-contract with Luna Innovations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Since the release of the first Field-Programmable Gate Array (FPGA) by Xilinx in 1985, FPGAs have significantly grown in size as well as complexity. Early FPGAs were fairly simple in the logic they could implement and were mainly used for hardware prototyping. Today, FPGAs can easily contain millions of gates and implement large scale hardware designs or Systems on a Chip (SoC).

One of the more interesting features of modern FPGAs is the ability to partially reconfigure a portion of the device while the rest remains operational. Partial reconfiguration has opened up a number of possibilities for hardware designers. Hardware updates can be made with little to no hardware downtime and allows the implementation of adaptive hardware that is self-modifying through partial reconfiguration. When configuring an FPGA, partial reconfiguration updates a subset of the configuration whereas a full reconfiguration updates the entire configuration, thus reducing the time to configure the FPGA.

While partial reconfiguration has it benefits, the currently available tools that support partial reconfiguration are limited in features and functionality. Xilinx provides two partial

reconfiguration flows that were initially targeted at the Virtex architecture: module-based and difference-based flows. The modular flow allows the designer to implement modular logic blocks that can be reconfigured with different modules, but requires a fairly strict set of guidelines to be met. The difference-based approach performs a bitwise difference between an older hardware design and an updated one and configures the differences between the two. This approach is typically used for small design changes only.

The thesis presents a tool known as the *Bitstream Manipulation Tool* or *BitMaT* that is part of an alternative module-based partial reconfiguration tool flow developed in the Configurable Computing Laboratory at Virginia Tech. When paired with the *wrapper generation tool*, the tool flow provides an easier means for module-based partial reconfiguration. BitMaT provides the low-level bit manipulation required to extract hardware modules and generate the appropriate partial bitstreams for reconfiguration.

1.2 Research Contributions

The BitMaT bitstream manipulation tool provides a critical element to a module-based partial reconfiguration tool flow. The flow addresses many of the shortcomings of the current mainstream approach. The goal of the tool flow is to provide an automated process to hardware designers as well as hide the low-level details required for successful module placement and routing.

Previous work has provided module-based partial reconfiguration flows, primarily targeting the older Xilinx Virtex architecture. BitMaT currently provides support for the Virtex-II and Virtex-II Pro architectures which are two of the more popular advanced FPGAs.

The alternative flow presented in this work opens up new opportunities for module-based partial reconfiguration. It allows module-based partial reconfiguration to more easily be used in target applications, and provides hardware engineers with tools that require only minimal restrictions on design.

1.3 Applications

Low-level modular extraction and placement is the main target application of BitMaT. Libraries of modules can easily be created using BitMaT as well as the generation of partial bitfile libraries for systems in which the hardware changes frequently.

Software defined radios (SDRs) provide an interesting application for the module-based design flow. An SDR uses programmable hardware to implement a radio. Radios are traditionally implemented in fixed hardware designs with fixed signal characteristics. SDRs on the other hand provide flexible signal characteristics. Using a module-based partial reconfiguration flow, each of the different signal characteristics can be implemented as an independent module which would allow for reconfiguration of the SDR to change the signal on the fly.

Another interesting application includes FPGA based single or multi-core SoCs or systems on a chip. Each CPU core could be attached to a reconfigurable module to provide coprocessor functions that could be swapped out depending on the CPU's current task. This work in addition to the rest of the alternative module-based flow would enable this application to be feasible.

1.4 Thesis Organization

This thesis is organized into the following chapters:

Chapter 2 - Background: This chapter reviews necessary topics such as FPGAs in general, Xilinx FPGAs and the common elements shared by various Xilinx families such as the Virtex-II and Virtex-II Pro architectures. Finally, a review of Xilinx's two flows for partial reconfiguration is presented.

Chapter 3 - Previous Work: This chapter begins with an overview of work in the area of partial reconfiguration followed by a section on the JBits API developed by Xilinx. Two

previous works on modular partial reconfiguration will be presented starting with PARBIT and then "Partially Reconfigurable Cores for Xilinx Virtex."

Chapter 4 - Tool Flow for Modular Based Partial Reconfiguration: This chapter begins by discussing the motivation for an alternative modular partial reconfiguration tool flow followed by the details of the proposed flow. BitMaT is an important component of the flow presented in this chapter. A discussion of the other tools involved provides an overview of the complete steps involved.

Chapter 5 - BitMaT: This chapter introduces BitMaT and its importance to the tool flow. Design objectives are listed followed by a discussion of how the objectives were met. Finally, information is provided on how to use the tool.

Chapter 6 - Results: This chapter presents the results of this work. A test design will be described as well as the results of the test. Tool performance and bitfile efficiency will be discussed and compared to the design objectives stated.

Chapter 7 - Conclusion: This chapter concludes the work by presenting a summary of the results and future work.

Chapter 2

Background

2.1 Overview

To understand BitMaT's function, it is necessary to review topics addressed in the thesis. The review begins with FPGAs, what they are and what they are capable of doing. Since BitMaT targets the Xilinx Virtex-II and Virtex-II Pro architectures, it is important to understand the general Xilinx FPGA architecture as well as specific capabilities of the Virtex-II and Virtex-II Pro FPGA architectures. Lastly, a review is given of the two partial reconfiguration flows as set forth by Xilinx.

2.2 FPGAs

FPGAs (Field-Programmable Gate Arrays) are semiconductor devices used for processing digital information. An FPGAs function is not fixed at the time of manufacture unlike ASIC (Application Specific Integrated Circuit) devices. FPGAs can be divided into two types: configurable and reconfigurable. Configurable FPGAs are one-time programmable, meaning, once the user has loaded a configuration, the function of the FPGA is fixed.

Reconfigurable FPGAs allow the device to be reconfigured multiple times.

Reconfiguration can be broken down into two sub-categories: static and dynamic [3]. Static reconfiguration implies that once the device is configured, it performs a specific function until it is fully reconfigured to perform a different function. Dynamic reconfiguration allows the user to change the hardware configuration while the current function is running, with minimal interruption of the logic.

Dynamic reconfiguration can occur fully or partially. Full dynamic reconfiguration entails sending a full configuration stream to the device. This is not the same as static reconfiguration because this method targets smaller hardware changes versus a complete hardware change. The downside of this method is, in the case of small hardware changes, you have an overhead of reconfiguring all logic in which no changes were made. Partial reconfiguration reduces the time overhead by allowing the user to configure the FPGA with a subset of the configuration bitstream.

There are a number of FPGA manufacturers including Actel [4], Altera [5], Atmel [6], Lattice Semiconductor [7], QuickLogic [8] and Xilinx [9]. Tools are usually provided from the manufacturer and allow the user to start with an HDL implementation of their hardware and through any number of steps, generate the necessary information to configure the FPGA. Typically, the tools will generate a serial bitstream that contains the necessary information to configure the function of the logic cells and routing within the FPGA. Figure 2.1 illustrates the Xilinx HDL design flow.

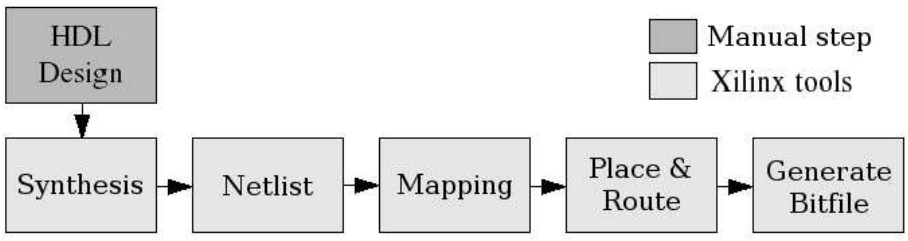

Figure 2.1: Xilinx HDL Hardware Design Flow

The flow begins with a manual step, HDL design. This step is where the hardware designer implements a design usually in either Verilog or VHDL. The synthesis step can be performed by the Xilinx synthesis tool (XST) or one of the other synthesis tools such as Synplify Pro from Cadence Design Systems [10]. The synthesis tool compiles and checks for errors in the design. A timing analysis is also performed to provide early feed back on whether the design will meet the timing limits of the resources. Once complete with no compiler errors, a netlist is generated and mapped to the device. The design is then placed and routed where the final timing analysis is performed. Lastly, if the previous steps returned no errors, a configuration bitstream is generated and written to a bitfile by the Bitgen program.

2.3 Xilinx FPGA Architecture

Modern Xilinx FPGAs are typically composed of the following: Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), Block RAMs (BRAMs), multipliers, and clock managers. Other configuration infrastructure includes Global Clocks (GCLKs), Input/Output Interfaces (IOIs), and BRAM Interconnect (BRAM_INT). All of the logic elements are connected by a network of complex routing and switch matrices as well as distributed clock resources. As an example, Figure 2.2 shows the architecture for the Xilinx Virtex-II XC2V80 device.

CLBs are the basic building blocks of user logic. They typically occupy the majority of the FPGA floorplan and are organized by rows and columns. The size of the device is primarily determined by the number of CLB rows and columns. For example, the XC2V80 in Figure 2.2 has 16 columns by 8 rows of CLBs and is one of the smaller Virtex-II devices. At the other extreme is the XC2V8000 with 104 columns by 112 rows.

A CLB consists of four slices. Figure 2.3, derived from Figure 2-1 in [1], shows an individual slice. Each slice contains two flip-flops (FFs) used for clocked logic, two, four-input lookup tables (LUTs) used for logic functions, and multiplexers. Each CLB contains a single

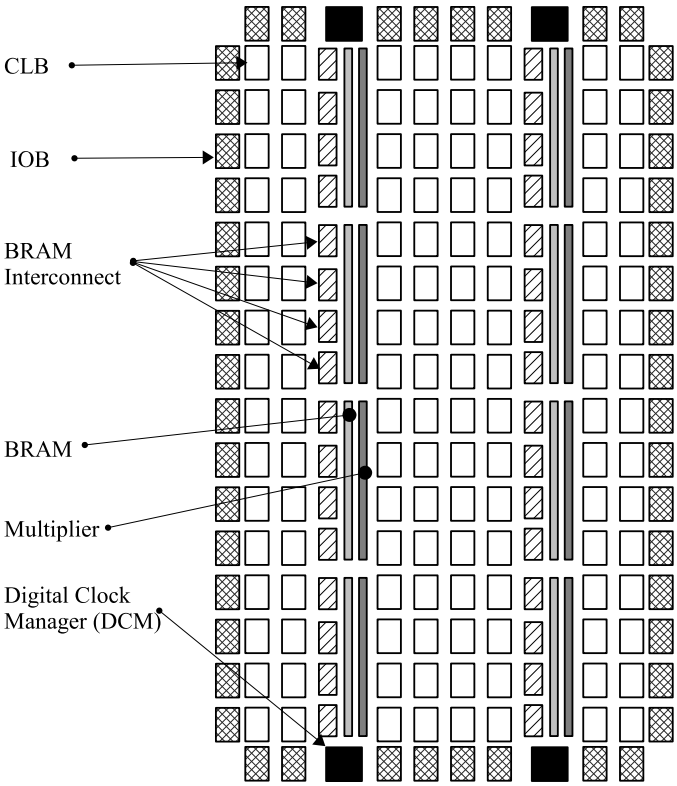

Figure 2.2: Xilinx XC2V80 Architecture

switch matrix which connects the external routing to the internal logic. IOBs contain the input/output logic for each pin on the device. The IOB signals can optionally be latched or registered. BRAMs provide memory buffers for user designs. The BRAM Interconnect (BRAM_INT) provides the routing interface to the BRAMs and the configuration for the dedicated multipliers which provide high speed 18x18 multiplication.

2.4 Xilinx Virtex-II/II Pro Configuration Architecture

The Virtex-II is a next generation architecture targeted at digital signal processing, networking and communications applications [1]. The Virtex-II Pro also incorporates up to two PowerPC 405 cores that are fully embedded within the FPGA as well as up to 20 RocketIO^{TM}

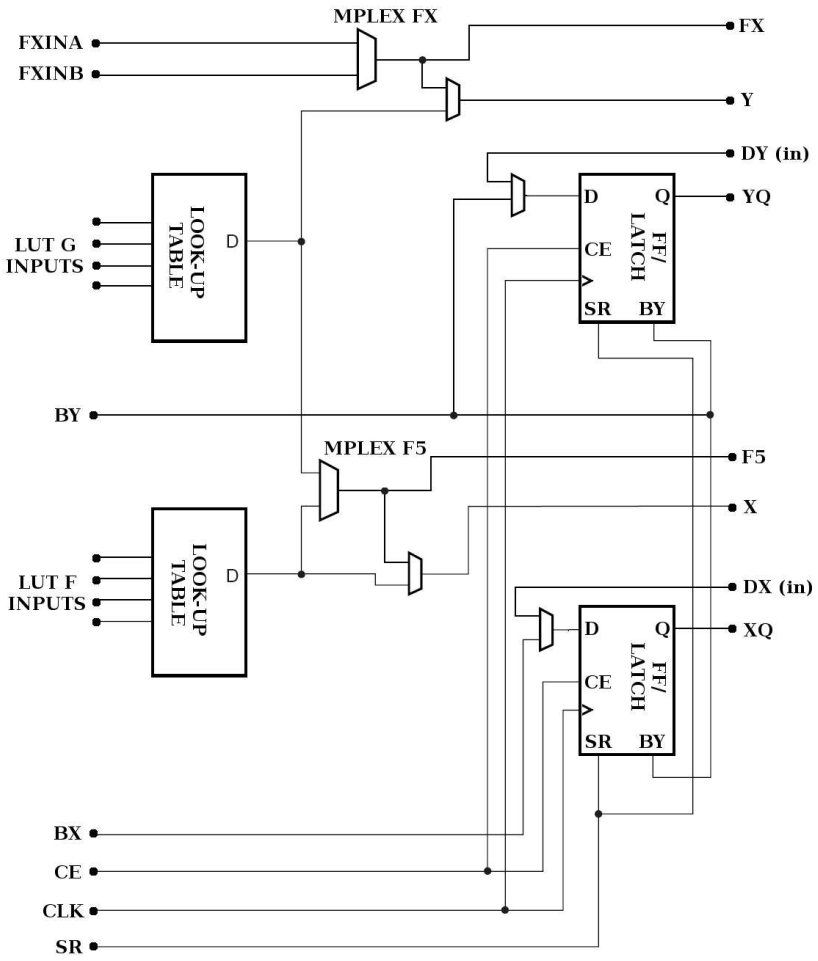

Figure 2.3: Virtex-II/Virtex-II Pro Slice

multi-gigabit transceivers(MGTs) [2]. The families contain a wide assortment of device sizes and features that fit many different applications.

2.4.1 Addressing

The smallest addressable element in a bitstream is called a frame. Frames are 1-bit in width and span vertically from the top to the bottom of the configuration memory. A frame is identified by a unique 32-bit address. The address composition is shown in Table 2.1.

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

Table 2.1: Frame Address Composition

The configuration address space is divided into three Block Addresses (BA). The first (BA 0) is the CLB block and contains all GCLK, IOB, IOI and CLB configuration frames. GCLK frames configure the global clock resources including clock buffers and digital clock managers (DCMs). IOB frames configure the I/O voltage standard while IOI frames configure the IOB registers, multiplexers and buffers.[1] CLB frames configure the logic blocks, routing and the majority of the interconnect.

The second (BA 1) is the BRAM block and contains the BRAM configuration. Finally, the third block (BA 2) is the BRAM Interconnect block and contains the BRAM Interconnect configuration bits. Each Block Address is further divided into column addresses or Major Addresses (MJA). Each Major Address addresses a specific configuration column within each Block Address as shown in Figure 2.4. For example, BA 0, MJA 0 would be the address for the GCLK configuration column in the configuration memory.

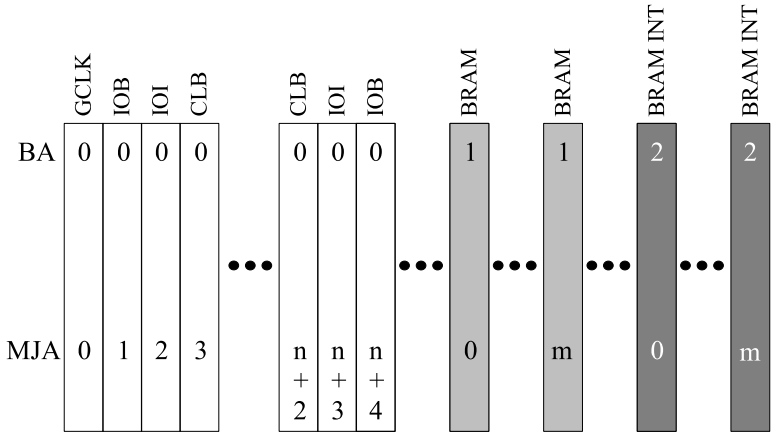

Figure 2.4: Column-Level Memory Map

[1]The IOB and IOI frames configure the left and right sides of the device. The top and bottom IOB/IOI configuration is contained in the top and bottom of the CLB configuration frames with which they are aligned.

The variable n in the figure corresponds to the number of CLB columns the device has and m corresponds to the number of BRAM and BRAM Interconnect columns. These variables differ with each device in both families and can be found in [1, 2]. The number of GCLK, IOB and IOI columns are the same for all Virtex-II and Virtex-II Pro devices. For each Block Address, the physical layout of the device corresponds to the MJA order in the configuration, meaning, the left-most CLB column is configured by the first (left-most) configuration column in the memory map. The GCLK column is a special case however because it is physically located in the center of the device, but shows up at the left-most position in the configuration.

The Minor Address (MNA) or Frame Address is used to address the individual frames within each Major Address. The number of frames varies depending upon the type of column and the length of the frame is variable between the different devices in the family.

2.4.2 Configuration

Configuration takes place by writing a bitfile to one of the configuration ports on the FPGA. Configuration can be loaded into the FPGA by three different methods: Master/Slave, SelectMap, and Boundary Scan.

Master/Slave and Boundary Scan are bit serial interfaces and SelectMAP is an 8-bit parallel interface. SelectMAP and Boundary Scan both support readback of the configuration memory. User logic can connect to the SelectMAP interface through the Internal Configuration Access Port (ICAP), allowing for self-reconfiguration.

The bitstream consists of a sequence of packets that contain header and data information for writing to the various configuration registers. There are 15 different configuration registers that can be used in the configuration process, as listed in Table 2.2.

There are two types of headers used for addressing the configuration registers. Type 1 packet headers, shown in Table 2.3, are used for packets up to $2^{11} - 1$ words in size, while a Type

Register Name	Address	Description
CRC	00000	CRC Register
FAR	00001	Frame Address Register
FDRI	00010	Frame Data Input Register
FDRO	00011	Frame Data Output Register
CMD	00100	Command Register
CTL	00101	Control Register
MASK	00110	Masking Register for CTL
STAT	00111	Status Register
LOUT	01000	Legacy Output Register (DOUT for daisy chain)
COR	01001	Configuration Options Register
MFWR	01010	Multiple Frame Write Register
FLR	01011	Frame Length Register
KEY	01100	Initial Key Address Register.
CBC	01101	Initial CBC Value Register
IDCODE	01110	Device ID Register

Table 2.2: Configuration Registers

2 packet header, shown in Table 2.4, should be used for packets containing up to $2^{27} - 1$ words.[2]

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

Table 2.3: Type 1 Packet Header

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

Table 2.4: Type 2 Packet Header

The following is an overview of the configuration registers for the Virtex-II and Virtex-II Pro FPGAs. Only registers related to the topic of this thesis will be described. Other register information is available in [1, 2].

[2]In the Xilinx configuration architecture a word is equivalent to 32-bits.

Cyclic Redundancy Check Register (CRC)

The Cyclic Redundancy Check Register is used to check data as it is written to any of the configuration registers. It is calculated using a 16-bit polynomial as shown in equation 2.1.

$$CRC - 16 = x^{16} + x^{15} + x^2 + 1 \tag{2.1}$$

A CRC check occurs at the end of each frame data register write (FDRI) using an Auto-CRC word or manually via the CRC command. The device checks the CRC by computing the check on the CRC word that was written; if the result is zero, the check passes, else the device goes into error mode and halts the current function. An example of a CRC packet is shown in Table 2.5.

30000001	**Write to CRC Register**
0000xxxx	**x = 16-bit CRC value**

Table 2.5: CRC Packet Example

Frame Address Register (FAR)

The Frame Address Register holds the configuration memory address to which frame data will be written. The FAR is auto-incremented when the number of words shifted into the Frame Data Input Register (FDRI) or out of the Frame Data Output Register (FDRO) reaches the value in the Frame Length Register (FLR). When reading or writing contiguous configuration data, the FAR is only loaded once with the starting address.

Each time the value in the FAR is changed, the command in the Command Register (CMD) is executed. An example of a FAR packet is shown in Table 2.6 with a Block Address of 0 (CLB Block), Major Address of 10 (CLB Column 7) and a Minor Address of 4 (4th frame

in column). The address composition is described in Table 2.1.

30002001	**Write to FAR Register**
00140800	**BA = 0, MJA = 10, MNA = 4**

Table 2.6: FAR Packet Example

Frame Data Input Register (FDRI)

Configuration data from the bitstream to be written to the configuration memory is shifted into the Frame Data Input Register. The location for which the data will be written is located at the address in the FAR. In order to successfully write to the configuration memory, the correct Device ID must have been written to the IDCODE register. Additionally, the Command Register (CMD) must be loaded with the Write Configuration (WCFG) command before writing to the FDRI.

The number of words written to the FDRI is specified in the header as explained before. The write operation is pipelined, meaning the first frame is written into configuration memory while the second frame is being loaded into the FDRI. This requires a single pad frame to be written at the end to flush the pipeline. After the number of words specified in the header have been written to the FDRI, an Auto-CRC word is written and the device automatically performs a cyclic redundancy check. Table 2.7 show an example FDRI packet.

30004xxx	**Type I FDRI, X = word count,** $X < 2^{11}$
....	X words of data
30004000	**Type II FDRI**
500xxxxx	**X = word count,** $2^{11} \leq X < 2^{27}$
....	X words of data

Table 2.7: FDRI Packet Example

Command	Code	Description
WCFG	0001	Write Configuration Data. Used with writes to the FDRI.
MFWR	0010	Multiple Frame Write. Used to write a single frame to multiple addresses.
LFRM	0011	Last Frame. De-asserts the GHIGH_B signal, activating all interconnect.
RCFG	0100	Read Configuration Data. Used with reads from the FDRO
START	0101	Begin Startup Sequence. Begins after a successful CRC check and DESYNC command.
RCAP	0110	Reset Capture. Resets the CAPTURE signal.
RCRC	0111	Reset CRC. Resets the CRC register.
AGHIGH	1000	Assert the GHIGH_B signal. Places all interconnect in a high-Z state to prevent contention when writing configuration data.
SWITCH	1001	Switch Configuration Clock (CCLK) Frequency as specified on the configuration option register (COR).
GRESTORE	1010	Pulse the GRESTORE signal. Sets/Resets IOB and CLB flip-flops.
SHUTDOWN	1011	Begin Shutdown Sequence. Activates on next RCRC command (typical) or CRC check.
GCAPTURE	1100	Pulse GCAPTURE. Loads capture cells with current register states.
DESYNCH	1101	Reset DALIGN Signal. Used at end of configuration to desynchronize the device.

Table 2.8: Command Register Codes

Command Register (CMD)

The Command Register holds a command that is used in the configuration logic to control global signals for specific configuration functions. After being written to the CMD register, the command is executed immediately. As mentioned before, the current command in the CMD register is executed again when an update on the FAR occurs. Table 2.9 shows an example CMD packet to execute the START sequence. Table 2.8 lists some of the possible commands.

30008001	**Write to CMD Register**
0000000A	**START Command**

Table 2.9: CMD Packet Example

Multiple Frame Write Register (MFWR)

The Multiple Frame Write Register is used when the compression option is enabled while running Bitgen. When more than one frame contains identical data, the frame can be loaded once into the MFWR and then written to multiple locations loaded into the FAR. The MFWR command is used with this option.

By utilizing Multiple Frame Writes, the size of the bitstream can be reduced according to the number of identical frames. An example multiple frame write sequence is shown in Figure 2.5.

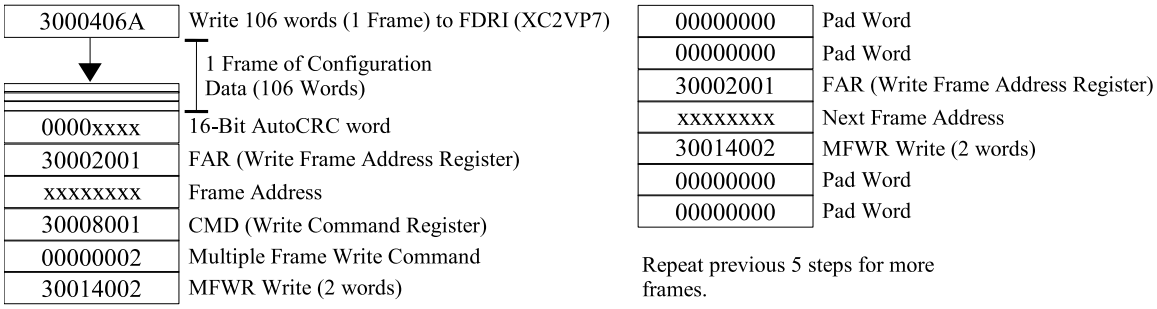

Figure 2.5: Multiple Frame Write Example

Frame Length Register (FLR)

The Frame Length Register holds one less than the length of a configuration frame for the device. This value must be loaded before any data is written to the device. This value is not hard-coded for the device because the same configuration data is used for all devices. Table 2.10 shows an example FLR write packet.

30016001	**Write to FLR Register**
xxxxxxxx	**X = Device Frame Length - 1**

Table 2.10: FLR Packet Example

Device ID Register (IDCODE)

The Device ID Register contains the device specific identification code. Writing the Device ID value to the IDCODE register will cause the configuration logic to automatically compare the two identification codes. This ensures that the bitstream to be loaded is for the correct device. If the values are not the same, the device goes into error mode and operation halts. A list of Device IDs is available in [1, 2]. Table 2.11 shows an example IDCODE write packet.

3001C001	**Write to IDCODE Register**
xxxxxxxx	**IDCODE (Device Specific)**

Table 2.11: FLR Packet Example

2.4.3 Bitstream Composition

Bitstreams for the Virtex-II and Virtex-II Pro consist of a bitstream header followed by configuration packets which consist of a header and data words as described in the previous subsection. There are two types of bitstreams: full and partial. Full bitstreams configure the entire configuration memory in the FPGA while partial bitstreams can configure a subset of the configuration memory.

The bitstream header is optional depending upon the tool that generated it. Bitgen usually includes a bitstream header for both partial and full bitstreams. The header contains information such as the Design Name (usually the name of the *.ncd* file), the Part Name, the Date and the Time. The format of the header is described in Table 2.12.

Length	Description	Value
2 bytes	Length Word	0x0009
9 bytes	Data Word(s)	0x0FF00FF00FF00FF000
2 bytes	Length Word	0x0001
1 byte	Design Name Token	0x61
2 bytes	Length Word	0x000B (variable)
10 bytes	Design Name String	example.ncd
1 byte	Part Name Token	0x62
2 bytes	Length Word	0x0009 (variable)
11 bytes	Part Name String	2vp7ff672
1 byte	Date Token	0x63
2 bytes	Length Word	0x000A
10 bytes	Date String	i.e. 2005/12/15
1 byte	Time Token	0x64
2 bytes	Length Word	0x0008
8 bytes	Time String	i.e. 12:01:23
1 byte	Bitstream Token	0x65
4 bytes	Length Word	Total bytes in Bitstream
Begin Bitstream		

Table 2.12: Bitstream Header Format

Full Bitstream Format

Figure 2.6 shows an example bitstream for the Virtex-II/II Pro FPGAs as generated by Bitgen.

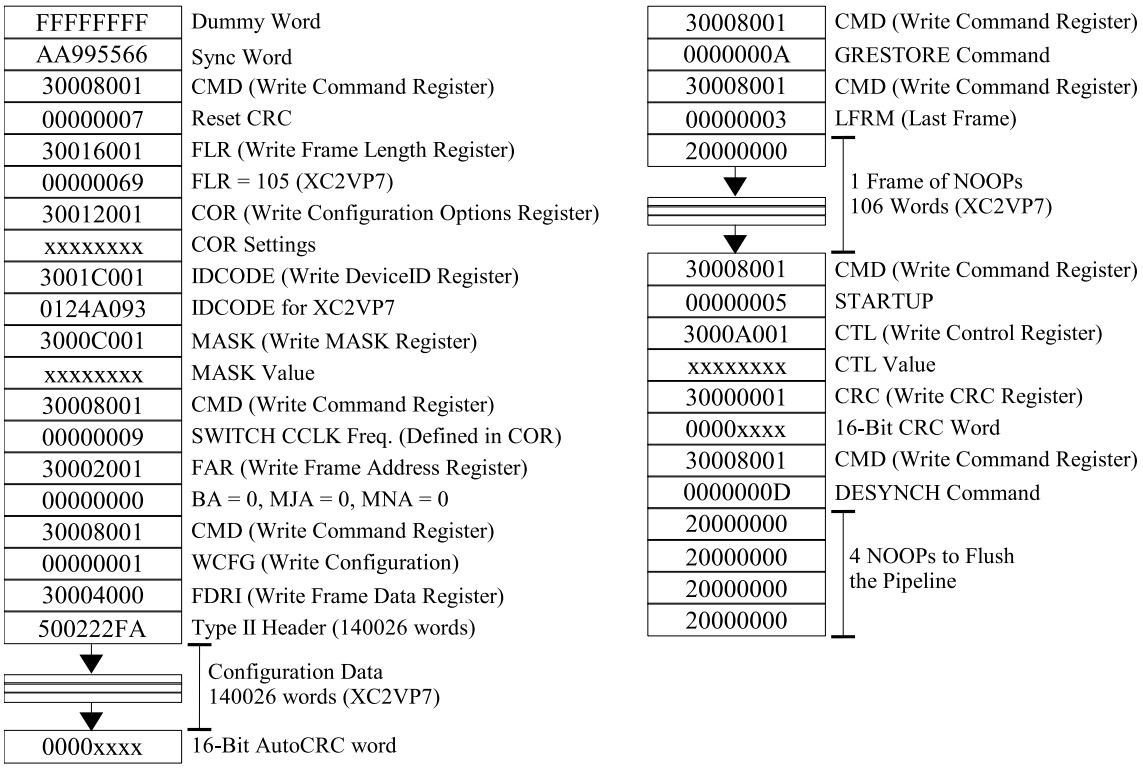

Figure 2.6: Virtex-II/II Pro Full Bitstream

Partial Bitstream Format

There are two types of partial bitstreams that can be generated: active and shutdown. Active partial reconfiguration will keep the device running during reconfiguration. This type of reconfiguration is useful when the static portion of the design needs to keep running. An example of an active partial bitstream is shown in Figure 2.7.

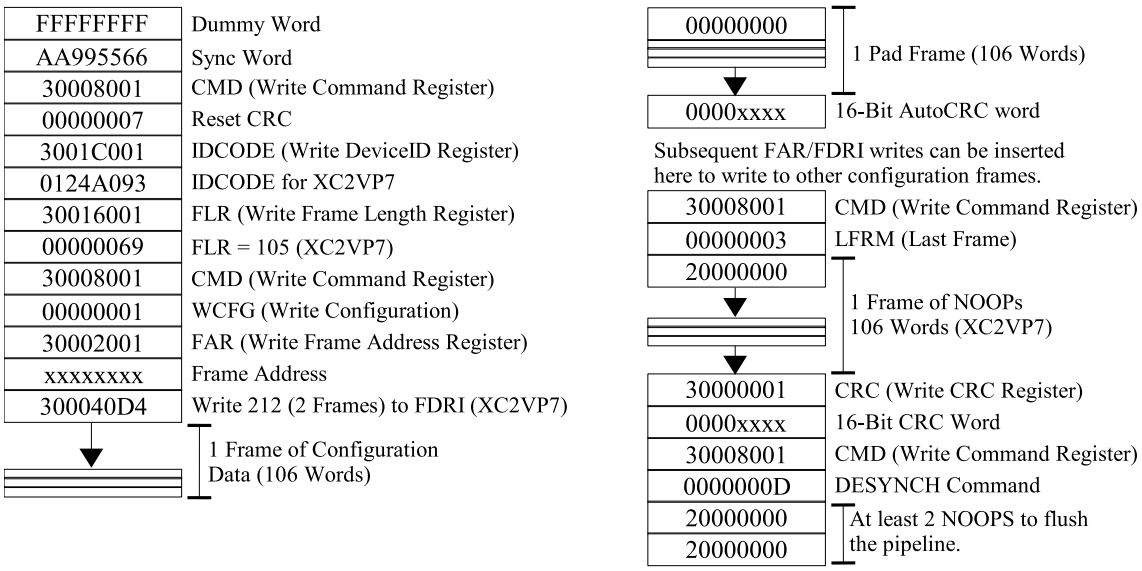

Figure 2.7: Virtex-II/II Pro Active Partial Bitstream

Shutdown partial reconfiguration simply halts the device before sending the configuration data. This can be used when operation of the static portion of the design is not necessary during reconfiguration. The ICAP port cannot be used in this method because its operation will be halted with the SHUTDOWN command. An example of a shutdown partial bitstream is shown in Figure 2.8.

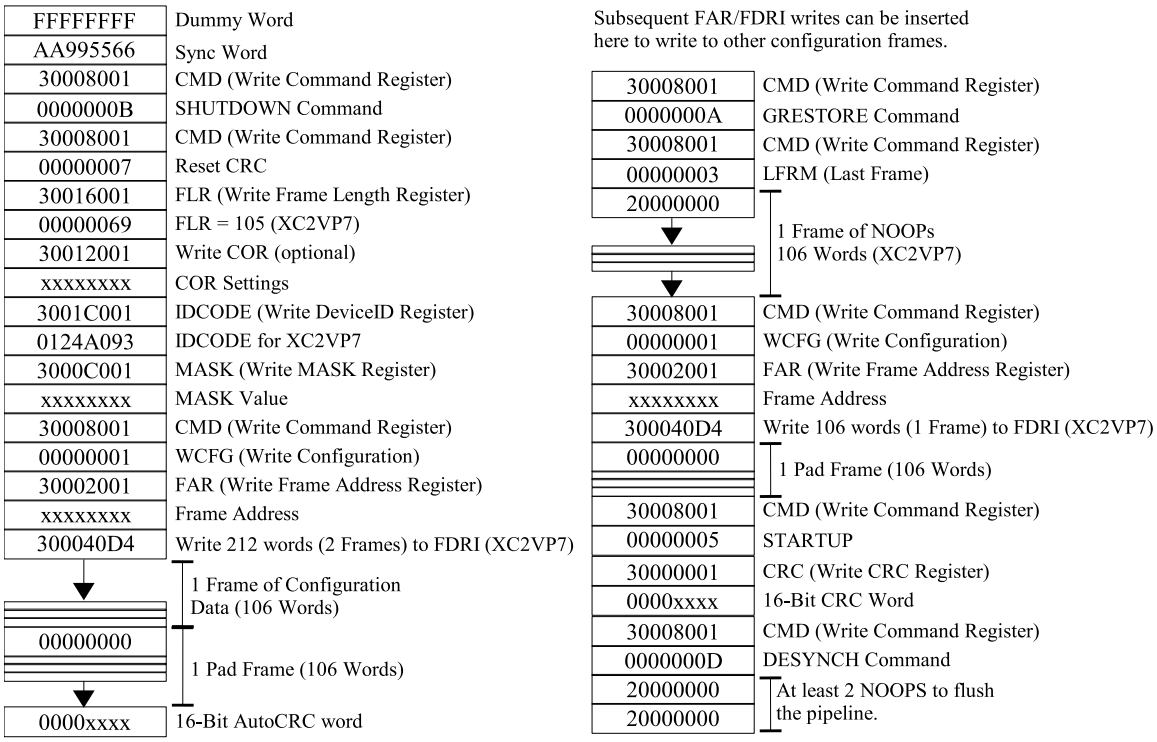

Figure 2.8: Virtex-II/II Pro Shutdown Partial Bitstream

2.5 Xilinx Partial Reconfiguration Flows

In [16], Xilinx presents two methods for partial reconfiguration of their Xilinx Virtex architecture: module-based and difference-based. While these flows were designed for the Virtex family, difference-based partial reconfiguration is a feature of the Xilinx tool *Bitgen* and can be used in any of the later architectures. Module-based is somewhat limited in the Virtex-II and Virtex-II Pro architectures and a different module-based flow will be implemented for the Virtex-4 architecture.

Partial reconfiguration can be accomplished in either of two configuration modes, SelectMAP mode or Boundary Scan (JTAG) mode. The smallest unit of reconfiguration is a single frame as discussed in the previous chapter.

2.5.1 Difference-Based Flow

The simpler of the two flows is difference-based partial reconfiguration. The user may use this method by making small changes to their design and using the "-r" switch when running Bitgen in the Xilinx tool flow. Using this switch, Bitgen simply performs a bitwise difference on the new and previous bitstreams. Any differences between the two are written into a partial bitstream as frames that need to be configured. When the partial bitstream is loaded onto the device, the changes made in the design will take effect.

This type of partial reconfiguration is very useful for small design changes and carries many benefits. It eliminates the need to reconfigure the entire device when perhaps less than one percent of the configuration is changing. Reconfiguring the entire device can be time-consuming compared to reconfiguring a small portion of the device. The partial bitfile itself is smaller in size, thus taking up less memory to store. Finally, in many cases the device will not have to stop functioning during this reconfiguration.

2.5.2 Module-Based Flow

Module-based reconfiguration requires that portions of the FPGA are designated as the "reconfigurable modules" while the rest of the configuration is static and operational. There are a number of size, communication and resource constraints associated with this method.

The size constraints require that:

1. The reconfigurable module must use the full height of the device.
2. The width must be a minimum of four slices and total width must be in four slice increments up to the entire width of the device.
3. The horizontal placement must also be on a four slice boundary.
4. The size of the module cannot be modified at a later time.

Communication between the module and the static logic must use a bus macro as defined in [16]. Each bus macro provides four bits of communication between modules and more than one bus macro may be used for a module. Bus macros may also be used for pass through connections. Pass throughs provide routes for signals that need to travel across a module that does not use the signal. A bus macro is composed of eight tri-state buffers (TBUFs) connected to bus lines. Communication between modules occures over the bus lines.

Resources considered to be a part of the reconfigurable module include all routing, TBUFs, BRAMs, Multipliers, and other logic contained within the boundaries. All IOBs that are immediately above or below, and in the case of left-most and right-most modules, the IOBs immediately to the left or right are included as resources contained within the module. Clocking resources are not included as they have their own configuration frames in the bitstream.

Figure 2.9 shows the design layout for Xilinx module-based partial reconfiguration.

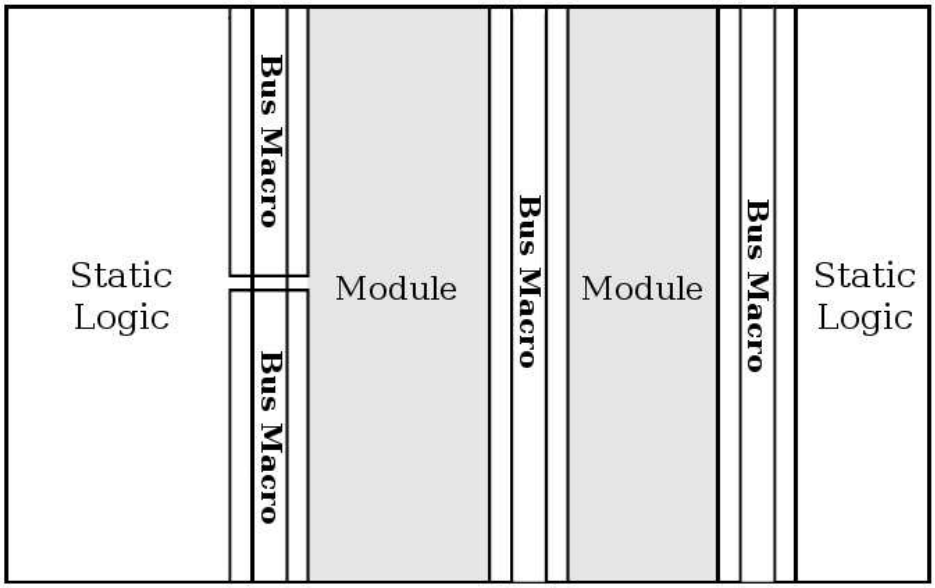

Figure 2.9: Xilinx module-based Design Layout

The grey areas show the reconfigurable areas. The reconfigurable modules are placed within

these types of regions. Communication between modules and static logic occurs across the bus macros.

Partial reconfiguration requires that the communication logic and wires must be fixed in the design. This ensures the connections do not change when the module is reconfigured. If the communication fabric is not static, reconfiguration will most likely result in floating wires which can produce indeterminate behavior.

The module-based design flow shares the size and configuration speed benefits as with the difference-based flow. Unlike the difference-based flow, the module-based flow allows for larger design changes by the ability to reconfigure a module to perform a different function entirely.

Chapter 3

Previous Work

3.1 Overview

While FPGAs have taken great advances in architectural features, use of these features has typically been plagued by lack of a good automated tool flow. As described in Section 2.5, the Xilinx modular partial reconfiguration flow has a strict set of rules a hardware designer must follow which can be time consuming.

This chapter discusses some of the efforts that have been put into developing new tool flows for partial reconfiguration. The discussion starts with the JBits application programming interface (API). JBits provides an alternative development environment to the standard HDL tool flow for static design. A key feature of JBits is it also provides access to the configuration bitstream.

Another low-level bitstream tool is PARBIT or PARtial BItfile Transformer. PARBIT targets the Virtex architecture and enables movement of module to different reconfigurable regions on the FPGA. Lastly, the work, "Partially Reconfigurable Cores for Xilinx Virtex" presents a novel design flow that utilizes a bitstream copy feature of JBits to reconfigure a coprocessor region for a hybrid CPU design.

3.2 JBits

The JBits Java API [13] provides a set of Java libraries that allows access to Xilinx FPGA bitstreams. Its purpose is to provide software support for dynamic reconfiguration of Xilinx FPGAs. JBits was designed for fast logic instantiation and compile times compared to HDL static design methods and its compile process. JBits version 2.x targets Xilinx Virtex FPGAs while the newest version, 3.0 provides support for Virtex-II FPGAs.

The JBits API was written in Java to be portable across a wide variety of systems. JBits works primarily by modifying the configurable resources of an FPGA, and operates on bitstreams generated by the Xilinx tool set or bitstreams read back from the FPGA through the configuration interface.

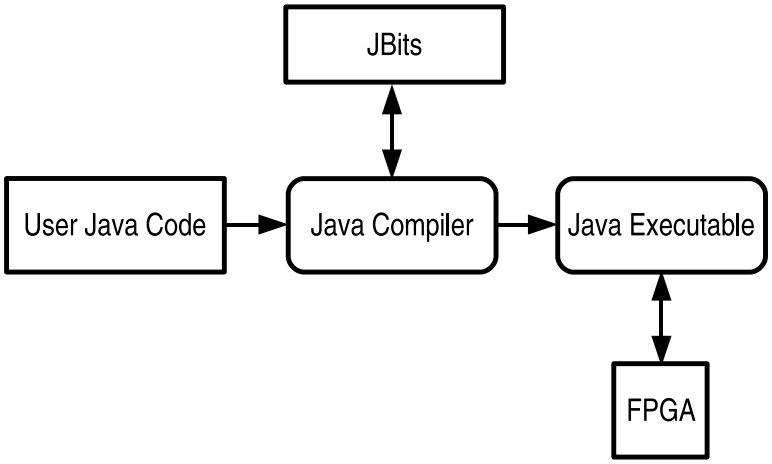

Figure 3.1: JBits Design Flow

The JBits design flow is shown in Figure 3.1. The Java libraries are pre-compiled Java classes that supply the user with parameterized executable code. This allows the user to generate dynamic bitstreams for the FPGA.

The user may also take a bitstream generated by the Xilinx tool flow and make changes to the design with JBits. JBits provides a difference-based partial reconfiguration flow by keeping track of design changes. Once the design changes are complete, the user may generate a

partial bitstream that will make the necessary updates to the FPGA configuration.

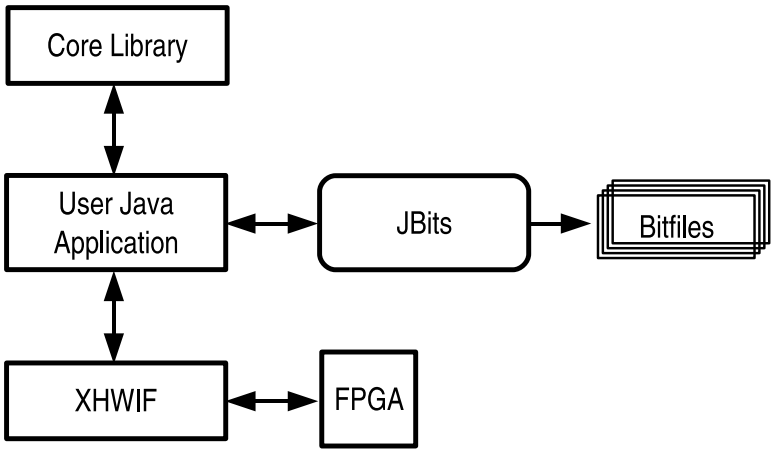

Figure 3.2: JBits System Design Flow

Figure 3.2 illustrates the JBits system design flow. The JBits interface provides a core library for user applications that contains counters, adders, multipliers and other standard logic. Additionally, the user may write Java code to instantiate logic to form their own cores for use in hardware designs. The XHWIF (Xilinx standard HardWare InterFace) block is an interface to download and readback configuration from the FPGA.

Drawbacks mentioned in [13] included the manual nature of JBits. Writing JBits code can be a tedious task because the developer must include code defining all elements of a design including the placement and routing. This drawback is somewhat alleviated by the use of logic cores mentioned before. JBits also requires the users to be very familiar with low-level details of the target FPGA architecture. These details are usually available from Xilinx, but most designers have not had the need to learn such information with standard hardware implementation flows. Finally, JBits provides no timing guarantees and there is a lack of analysis tools due to the level that JBits works. The development of BoardScope [14] has helped to provide one solution for design analysis at the lower level.

3.3 PARBIT

PARBIT (PARtial BItfile Transformer) transforms Xilinx Virtex configuration bitfiles to enable Dynamically loadable Hardware Plug-in (DHP) modules [11]. The tool enables the hardware developer to define and configure a reconfigurable region on the FPGA.

There are two modes of operation: slice and block mode. In slice mode the user defines one or more full CLB columns as the slice from the original bitstream. The tool then generates a partial bitstream containing the slice in the same position as in the original bitstream. Block mode allows the user to define a block of CLBs in the original bitstream and generate a partial bitstream with the block in a different location. Options include specifying whether the chip will be shutdown prior to reconfiguration, to selecting between the JTAG and SelectMAP configuration ports for loading the bitstream, and specifying the side (left or right) of the chip to configure while in slice mode.

Block mode provides movement of a target block of CLBs and requires start/end columns, start/end rows and a target row/column as inputs to PARBIT. Movement of a module requires boundary connections to be implemented in the new location or else relocation is ineffectual if the global logic cannot communicate with the relocated module. Connections to the module are to be defined by the user. In [12] a *gasket* interface is used to connect the modules to the global logic which provide fixed routing of signals from outside logic to the DHP.

PARBIT proves to be a useful solution for partial reconfiguration of Virtex FPGAs. It enables the user to transform and combine bitstreams for instantiation and movement of dynamically loadable logic.

3.4 Partially Reconfigurable Cores for Xilinx Virtex

The work in [15] focuses on developing a design flow for the implementation of hybrid processors on Virtex FPGAs and was motivated from the lack of high-level design tools to perform partial reconfiguration. Hybrid cores are CPUs coupled with configurable logic in an attempt to overcome interface limitations. Partial reconfiguration of hybrid CPUs is beneficial because it allows for modification of the interface between the CPU core and configurable logic. For soft CPU cores, the core itself may be modified.

For the same reasons mentioned in the module-based partial reconfiguration flow, communication interfaces must be static between the static and dynamic portions of the design. Static routes passing through the reconfigurable region must also be dealt with or they provide the potential to be broken during reconfiguration if the route is not define in the new configuration. They solved the interface and routing problems developing two techniques, the virtual socket and feed-through components.

Figure 3.4 from [15] illustrates the use of the virtual socket technique to reconfigure co-processor #2 in place of co-processor #1. Virtual sockets provide the interface between the reconfigurable coprocessor core and the static portion of the design. JBits is used to copy the coprocessor core from a template-like bitfile and generate a partial bitstream that will configure the device with the coprocessor in place.

Both the template and the target bitstreams contain the virtual socket interface, thus making a seamless connection to the newly configured coprocessor core. As mentioned earlier, long static routes have a tendency to be routed through the reconfigurable region. When the region is reconfigured the route is lost, disconnecting any signal along that path. Figure 3.3 shows a before and after illustration of the feed-through process.

To solve this problem, the use of a feed-through component causes the router to locate the signal around the dynamic region. The component itself is just an anchor point that simply passes the signal along.

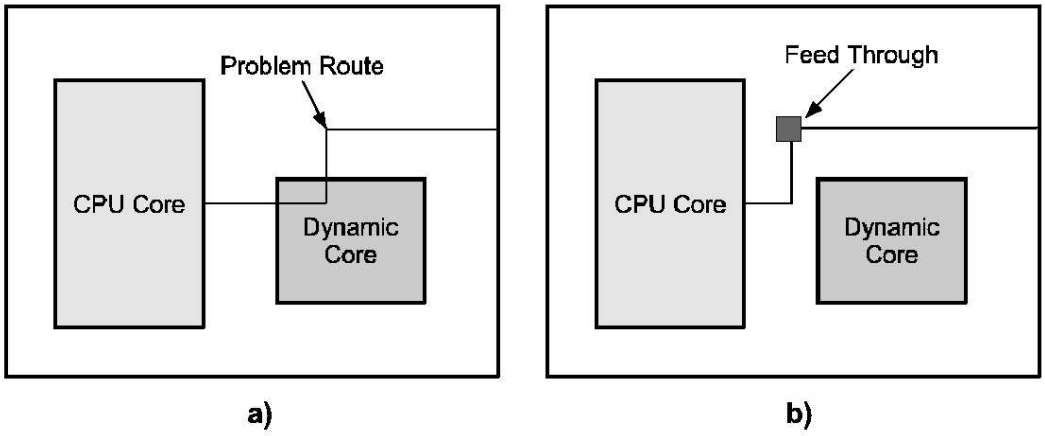

Figure 3.3: Feed Through Component Example

The tool flow is composed of two separate parts. The first flow describes the design process of the CPU core and the second for the coprocessor design. Figure 3.4, derived from Figure 2 in [15], illustrates the virtual socket based flow.

1. Create a full design that includes the CPU core and a coprocessor (Figure 3.4(a)).
2. Connect the CPU and the coprocessor with a predefined virtual socket interface component (Figure 3.4(a)).
3. Using constraints on the placement of the CPU core, coprocessor and virtual socket, run the tools to generate the initial full bitstream (Figure 3.4(b)).

Figures 3.4 (b), (c) and (d) illustrate the second flow and are described in the following steps:

1. Create a full design with a coprocessor (Figure 3.4(c)).
2. Connect the coprocessor to a virtual socket and connect the static side of the virtual socket to unused I/O pins to avoid having the router optimize the socket away (Figure 3.4(c)).

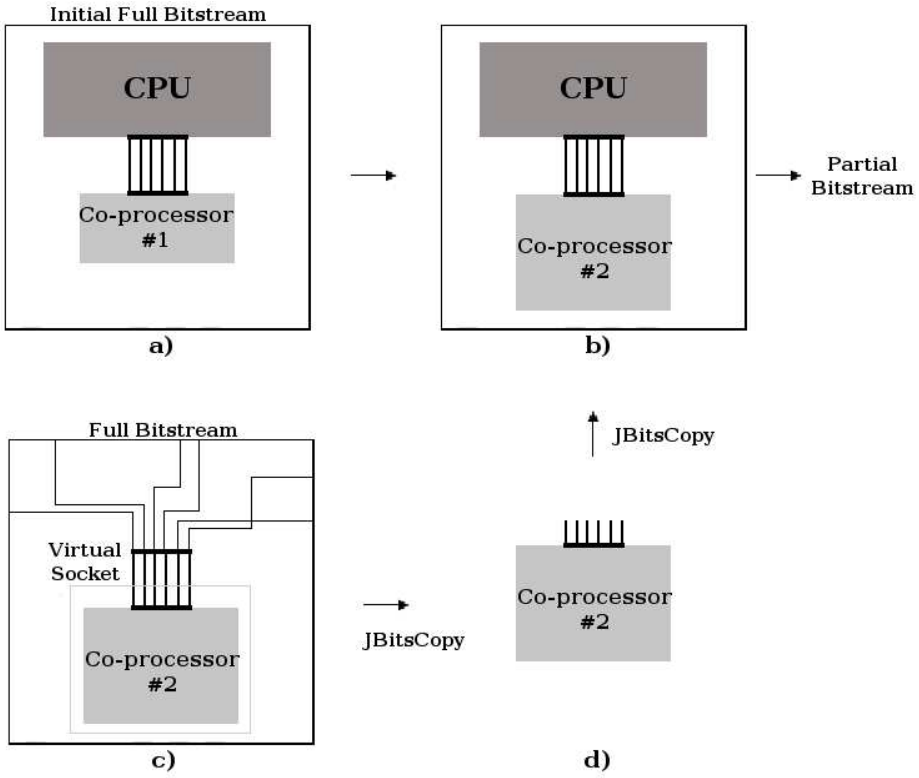

Figure 3.4: Virtual Socket Based Design

3. Using constraints on the placement of the coprocessor and virtual socket, run the tools to generate the full bitstream.

4. Extract the coprocessor and virtual socket (Figure 3.4(d)), and merge the coprocessor core into the initial design (Figure 3.4(b)).

This design approach requires some work on the designer's part, but it provides flexibility that the current mainstream tools do not offer. This particular method has been successfully tested on an audio streaming applications as described in [15].

Chapter 4

Tool Flow for Module-Based Partial Reconfiguration

4.1 Overview

The basis for the design and implementation of BitMaT stems from the work on the Dynamic Module Server (DMS) in the Configurable Computing Lab at Virginia Tech. The DMS requires an automated, module-based partial reconfiguration tool flow. However, current industry tools have not been able to provide an acceptable tool flow for modular partial reconfiguration.

This chapter presents the design objectives of the tool chain and the proposed flow for modular partial reconfiguration within the Dynamic Module Server project.

4.2 Design Objectives

The proposed modular partial reconfiguration tool flow had a number of objectives to meet:

1. Constrain routing resources.
2. Develop a module communication interface.
3. Automation, require very little designer intervention.
4. Target the Virtex-II and Virtex-II Pro architectures.

Previous work on modular flows has provided some help with these objectives, but they do not support the Virtex-II or Virtex-II Pro architectures which is the initial target architecture for the DMS. Both the PARBIT [11] flow and the hybrid CPU flows [15] support only the Virtex architecture.

Routing resources are in abundance within the FPGA fabric. Routing algorithms can be very complex and in the mainstream tools critical paths get priority. Timing-driven routing is important for a modular partial reconfiguration flow to be successful.

For a module to successfully work when configured, the communication interface from that module needs to be connected to the proper static paths. Routing problems that were presented in [15] and [11] are still of a concern and must be solved. The communication interface is also important. Modules need to have an interface that will provide communication between the static and dynamic regions. Static placement of the communication interfaces must line up correctly during partial reconfiguration.

4.3 Proposed Flow

During the work on the DMS project, all of the objectives were addressed and met although the routing solution is still under development. Figure 4.1 illustrates the proposed design flow for modular partial reconfiguration.

The design flow for the Dynamic Module Server project begins with HDL design which has been floorplanned into static and dynamic regions. Floorplanning can be achieved by

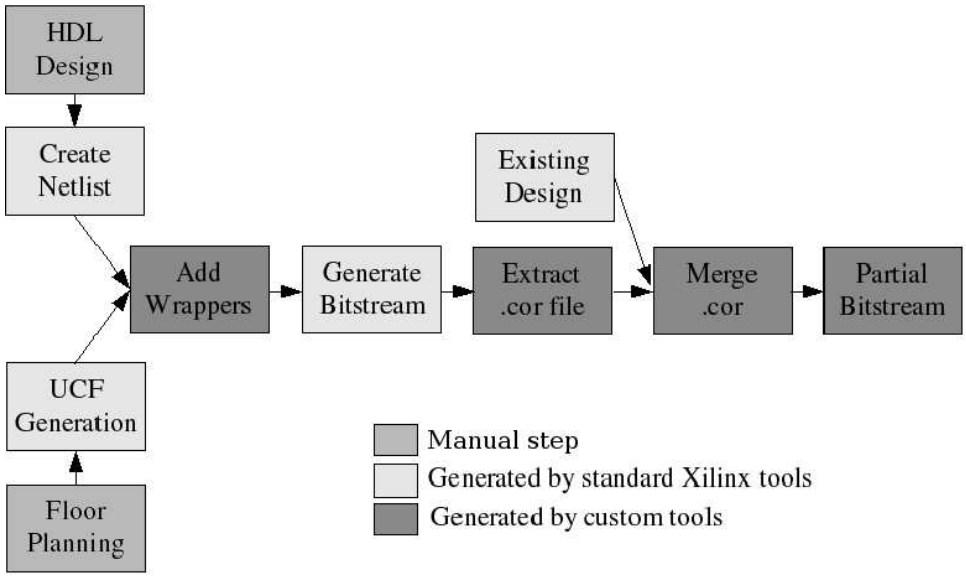

Figure 4.1: DMS Modular Partial Reconfiguration Tool Flow

either using the graphical Xilinx Floorplanner tool, or manually by defining a constraint file containing area constraints.

Using the Xilinx tools, a netlist is generated. The wrapper generation tool is then responsible for determining the connections to the partial module(s) in the user design and generates HDL to implement the wrapper(s). Once the wrappers are generated, a full bitstream is generated using the Xilinx tools. The wrapper generation tool eliminates the need to hand place the connections to the dynamic region(s).

With the modules in place and wrapped, BitMaT will extract the configuration for the module(s) in the full bitstream and save them in a *.cor* file. Repeating steps 1 - 5 will allow the user to define different modules in the dynamic region(s) and generate the corresponding *.cor* files (described in the next chapter). In the last two steps, BitMaT generates a partial bitstream containing any one of the modules created in the previous step. This is done by taking the initial existing full bitstream and merging a *.cor* file into the column. These steps are repeated for each of the *.cor* files.

4.3.1 Wrapper Generation

The wrappers provide connections to the static logic using standard logic and routing resources. The Xilinx module-based flow implements these connections as bus macros that use TBUFs, which are in limited supply for Virtex and Virtex-II/II Pro architectures and are unavailable in the Virtex-4 architecture. Using standard logic and routing resources for communication provides more flexibility as well as density. Each bus macro provides only four bits of intermodule communication. An example illustrating the function of the wrapper generation tool is shown in Figure 4.2.

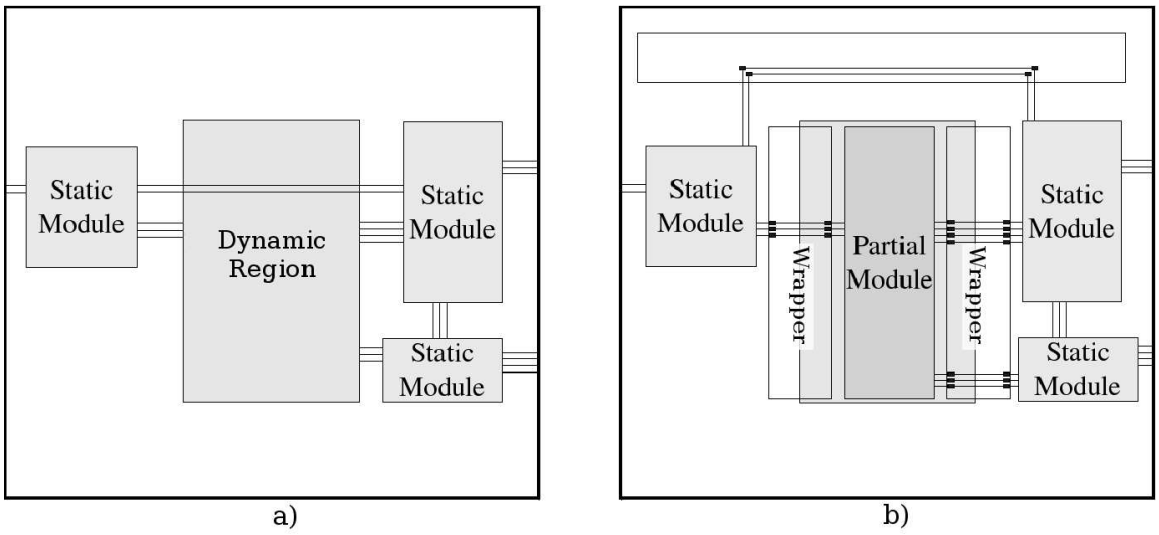

Figure 4.2: Wrapper Generation

Before the wrapper generation tool is run, the wires are directly connected to the logic within the dynamic region. Without the wrapper, changes to the logic within the dynamic region will cause the wires to be routed differently. This poses a problem with partial reconfiguration because connections into a module will not be preserved if they do not have the same entry point when reconfiguration takes place.

The same problem occurs for static module to static module connections. In Figure 4.2 (a), two wires can be observed crossing the dynamic region at the top. During reconfiguration,

those routes will most likely be invalidated in the new configuration for the region.

Figure 4.2 (b) shows the design after the wrapper generation. In this case, there is a wrapper on either side of the dynamic region. For each connection there are two anchor points for the route. The anchor points ensure that each partial module will be connected to the wrapper interface when partially reconfigured.

Note that the two static module to static modules routes at the top have now been re-routed. By specifying anchor points in strategic locations, the user may force the route to go around the dynamic region. A routing tool such as ADB [17] could be used with the wrapper generation tool to handle the problem routes.

4.3.2 Bitstream Manipulation

BitMaT provides the bitstream manipulation interface. The Dynamic Module Server will contain a library of modules that clients request. BitMaT provides the initial library of modules and can be called by the DMS to generate a new module that is not currently in the library.

BitMaT depends upon the wrapper generation tool to take care of all interface requirements. The tool then takes an original bitstream (full or partial) and uses it as the static logic and routing template. This ensures that any static logic that may be reconfigured along with the partial module will be maintained and glitch free. Once the original bitstream is read in, a *.cor* file is read and placed in the dynamic portion of the configuration bitstream. Finally a partial bitstream with the new module in place is generated. The bitstream manipulation process is further detailed in Chapter 5.

Chapter 5

BitMaT

BitMaT is a C program that was developed to perform configuration extraction on FPGA bitfiles as well as partial bitfile generation. The considerable complexity of bitstreams has made the development of such a tool a non-trivial endeavor. Some of the larger Virtex-II Pro devices contain 25-35 million configuration bits. Developement of BitMaT required many issues to be solved with very little documentation. While some information is available about the configuration architecture such as described in Chapter 2, other information had to be extracted from tedious analysis of various configuration bitstreams.

This chapter discusses the design objectives that went into the development of BitMaT, followed by the program flow, implementation and usage.

5.1 Design Objectives

A number of design objectives were required for this work to be successful as well as useful:

- A simple command-line interface.
- Runtime efficiency and performance.

- Portable to different environments.
- Efficient and functional partial bitstream generation.

A simple command-line interface is important not only for ease of use but particularly for use in an automated system. The Dynamic Module Server will need to make calls to the tool set on the fly so a simple command line interface will make its deployment easier.

Runtime efficiency and performance are objectives that are quite common in any tool development. In this case, the automated deployment of this tool requires the operations to be performed quickly and with minimal memory usage. Portability is also important because the DMS could potentially be run in an embedded environment.

Finally, efficient and functional partial bitstream generation is important for several reasons. The partial bitstreams must contain the correct set of configuration frames as specified by the user. All configuration pertaining to the function of the module must be included for the reconfiguration to be successful. Efficiency is the main objective of partial reconfiguration. The use of configuration short cuts such as multiple frames writes can reduce the time to configure the device as well as the space requirements for storing the configuration files.

5.2 Implementation

5.2.1 Language Selection

To effectively extract configuration and generate partial bitstreams, it was necessary to have an interface to the Xilinx bitfile format. Initially, JBits seemed to provide an easy solution to this problem since it provided a direct interface to Xilinx bitstreams. It was later realized that a C implementation would be more efficient than the Java based JBits approach. The C implementation only required the additional development of a bitfile parser. Java's ability to run on multiple environments was another one of the benefits for JBits, however, there

are many tools and compilers for C that provide portability across different systems.

5.2.2 Data Organization

Two types of C structures were created to store information: a device structure and a bitstream structure. The bitstream structure contains member variables that hold the bitfile header information and the original bitstream data. This structure is initialized during the bitfile parsing process as the values are read. The following code shows the *bitstream* C structure:

```
struct bitstream {
        char * design_name;       // Design Name
        int design_name_len;      // Length of Design Name
        char * part_name;         // Part Name
        int part_name_len;        // Length of Part Name
        char * date;              // Date
        int date_len;             // Length of Date
        char * time;              // Time
        int time_len;             // Length of Time
        unsigned char * data;     // Bitstream Data
        int data_len;             // Length of Bitstream Data
};
```

The second C structure is the device structure. The device structure maintains a large number of member variables that contain information about the device, from the number of rows and columns the device contains to the number of frames in each of the column types. The device structure also contains a configuration data array that is used to build the partial bitstreams. The organization of the bitstream in the array is exactly the same as it is in the configuration bitstream. A special function was written to convert between Block, Major

and Minor addresses to the index within the array.

In order to extract configuration bits from a range of CLB rows, it was necessary to decompose the frame structure into rows. As mentioned in the overview, an analysis of numerous test bitstreams was required. For the XC2V1000 FPGA there are 40 CLB rows and 106 32-bit words per frame (424 bytes). There are three words of IOB/GCLK configuration bits at the top and bottom of the frame. This leaves $424 - ((3\ words * 4\ bytes/word) * 2)$ or 400 CLB configuration bytes. Divide that by 40 rows and there are 10 bytes per CLB row. Table 5.1 shows the frame structure for the XC2V1000 FPGA.

Byte Position	**Description**
0 - 11	Top IOB/GCLK Configuration
12 - 21	Top CLB Row
22 - 31	CLB
...	CLB
...	CLB
392 - 401	CLB
402 - 411	Bottom CLB Row
412 - 423	Bottom IOB/GCLK Configuration

Table 5.1: Configuration Frame Structure

5.3 Program Flow

BitMaT executes in two main modes of operation, extraction mode and merge mode. Figure 5.1 shows the program flow for BitMaT.

First the user inputs are checked for correctness; if there is an error, a message and usage summary are displayed. Next the target bitstream is parsed. The input bitstream may either be a full bitstream or partial bitstream. The original implementation of BitMaT only supported full bitstream input, but later partial bitstream input was a feature added to allow for multiple modules to be placed within the same column on a device. Placement of multiple modules in the same column is achieved by taking a partial bitfile previously

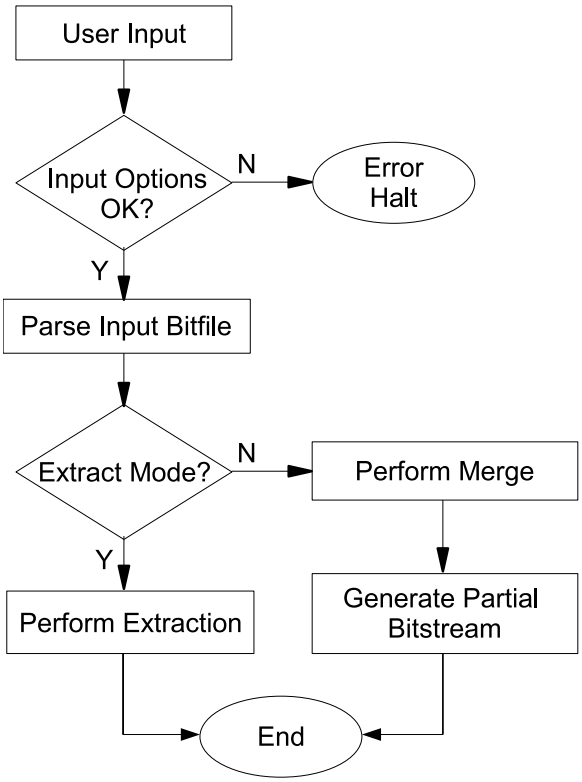

Figure 5.1: BitMaT Flow

generated by BitMaT back through the tool with a new target module to be placed in the same column.

Next the program branches into either extraction or merge mode. Each of these modes manipulate the bitstream by either extracting configuration bits or by merging static configuration with a new module to generate a partial bitstream.

These modes are described in greater detail in the following sections. Once either mode completes, the program ends. Generally, the overall structure of BitMaT is fairly efficient and the computational complexity is linear in the size of the target reconfigurable region.

5.3.1 Extraction Mode

In extraction mode, BitMaT accepts a bitstream that contains one or more wrapped partial modules, two sets of coordinates, and an output filename. The first set of coordinates describes the target block of CLB configuration, while the second describes the target block of BRAM configuration. The target BRAM Interconnect configuration coordinates are derived later from the CLB coordinates.

Before BitMaT begins the extraction process, it performs a CRC on the full bitstream and compares the computed checksums with the checksums written by *Bitgen*. This check is to ensure the configuration data is not corrupt. Once the check completes and is successful, the extraction process continues. Table 5.2 shows the structure of the *.cor* file generated by the extraction process.

32-bit Word (Hex)	**Description**
Device IDCODE	IDCODE to verify part
00000000	Block Address 0 (CLB block)
xxxxyyyy	x = CLB Start Column, y = CLB End Column
xxxxyyyy	x = CLB Start Row, y = CLB End Row
....	All GCLK Configuration Frames
....	Target Configuration CLB Frames
00000001	Block Address 1 (BRAM Block)
xxxxyyyy	x = BRAM Start Column, y = BRAM End Column
xxxxyyyy	x = BRAM Start Row, y = BRAM End Row
....	Target BRAM Configuration Frames
00000002	Block Address 2 (BRAM_INT Block)
xxxxyyyy	x = BRAM_INT Start Column, y = BRAM_INT End Column
xxxxyyyy	x = BRAM_INT Start Row, y = BRAM_INT End Row
....	Target BRAM_INT Configuration Frames
0xFFFFFFFF	Dummy Word, End of File

Table 5.2: .cor File Format

The extraction process begins by writing the IDCODE of the target device to the *.cor* file. This prevents designs targeting different devices from being mismatched when merging modules with bitstreams. Next the block address of the configuration is written to the *.cor*

file.

The first is Block Address 0 (CLB). The first set of frames to store are the GCLK configuration frames, the GCLK configuration frames contain information for enabling the clock domains within the device. This information is necessary to ensure proper function of the module and all GCLK frames are written regardless the size and location of the dynamic region. The target CLB configuration frames are written next.

For Block Addresses 1 (BRAM) and 2 (BRAM_INT), the process is similar to the previous steps except for the GCLK configuration frames. In some cases, the BRAM and BRAM_INT blocks may not be targeted and thus will be excluded from the *.cor* file.

Prior to the extraction of the BRAM interconnect, the target coordinates must be computed. For Virtex-II Pro devices, this computation is trivial as the BRAM Interconnect columns are evenly spaced across the device. Virtex-II devices require a slightly different computation as the spacing is not consistent across the entire device, but divided at the center with even spacing to the left and right of center.

Once the target configuration words have been written a final dummy word is written to indicate the end of the file. Figure 5.2 show a graphical representation of the extraction process.

5.3.2 Merge Mode

In merge mode, the tool takes a full bitstream of the current design and a *.cor* file containing a module to be placed, and generates a partial bitstream with the new module in place. The partial bitstream generated also contains the configuration bits from above and below the module containing information about routing critical to the function of other parts of the design.

Figure 5.3 illustrates the merge process as it will be described. The merge process begins by parsing the selected *.cor* file into a target bitstream array. The current bitstream (current

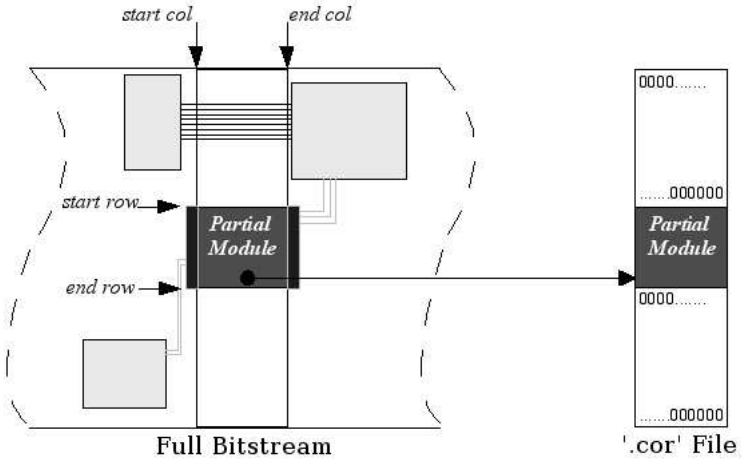

Figure 5.2: Module Extraction Mode

design) information is then merged with the configuration information that was previously store in the *.cor* file.

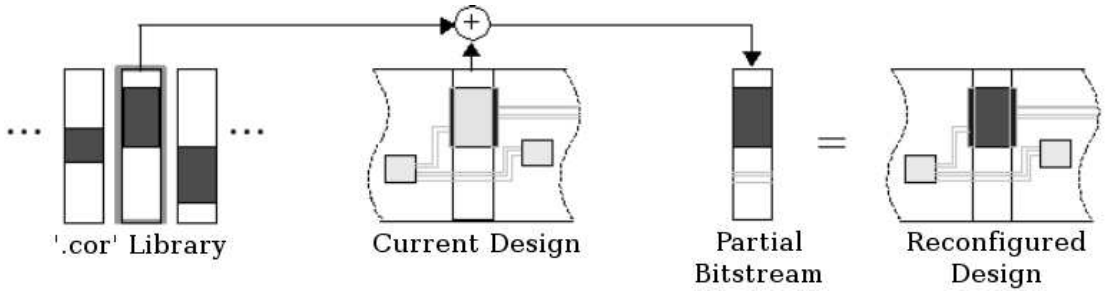

Figure 5.3: Module Merge Mode

The merge involves removing frames from the current design equal to the width of the module being extracted. This width must also be identical to the width of the new module being placed in the design. Placement of the new module occurs by simply overwriting the configuration bits of the region occupied by the new module. This places the new module into the original design, leaving surrounding configuration bits untouched. In Figure 5.3 the static routes below the partial region were preserved in the reconfigured design.

The GCLK frames that were accompanying the new module take a different path. Instead

of replacing the frames in the current design, a bitwise OR is performed on the two sets of GCLK frames. This effectively enables any clock resources used by the new module without modifying clock resources that might be used by the static portions of the design. The ORing operation is also performed on the three pad words at the top and bottom of each frame to account for the additional GCLK bits.

Next, an optimized bitstream is generated by writing all zero frames using the multiple frame write command. This command, as described in Section 2.4.2, is much faster for writing identical frames because the bitstream only loads the configuration frame once and subsequently writes the frame to multiple addresses. Significant configuration time can be saved by utilizing multiple frame writes.

Once all of the multiple frame write commands have been written, the non-zero frames are written using the partial bitstream format in Figure 2.7. BitMaT computes new CRC values for the partial bitstream using the CRC algorithm described in Section 2.4.2.

The configuration frame write order is slightly different than the standard CLB, BRAM, BRAM_INT sequence observed in Bitgen generate bitfiles. Results show that it was better to reconfigure the BRAMs after the BRAM_INTs. During the BRAM_INT reconfiguration, the inputs to the BRAMs toggle which causes data corruption in the BRAMs. Once the partial bitstream has been written, the merge is complete.

5.4 Usage

BitMaT currently has a very simple command line interface. There are two basic options *-e* for extraction mode and *-m* for merge mode. Each of the options has a set of required command line inputs that pertain to the mode of operation. The following is a general usage of BitMaT:

```
bitmat -e <device> <input bitstream> <area file> <output filename>
```

or

```
bitmat -m <device> <input bitstream> <.cor file> <output filename>
```

For extraction mode, *<input bitstream>* is the bitfile for which the extraction will be performed on. *<area file>* is an automatically generated file from the wrapper generation tool that defines the location of the dynamic region. *<output filename>* is the name of the *.cor* file for storing the extracted module.

For merge mode, *<input bitstream>* is the bitfile that is to maintain static logic in the partial bitfile. *<.cor file>* contains the module to be partially reconfigured into the device, and *<output filename>* is the name of the partial bitfile that will be generated.

The currently supported Xilinx FPGAs are:

Virtex-II: XC2V40, XC2V80, XC2V250, XC2V500, XC2V1000, XC2V1500, XC2V2000, XC2V3000, XC2V4000, XC2V6000 and XC2V8000

Virtex-II Pro: XC2VP2, XC2VP4, XC2VP7, XC2VP20, XC2VP30, XC2VP40, XC2VP50, XC2VP70 and XC2VP100

Chapter 6

Results

6.1 Overview

Demonstrations of the current version of the modular tool flow have shown successful results. This chapter presents a test design using the proposed module-based partial reconfiguration flow and the results. Runtime performance and efficiency as well as bitstream generation will be presented to support the original design objectives listed in Section 5.1.

6.2 Test Design

The test platform used was an ML300 prototyping board with a Virtex-II Pro XC2VP7 FPGA and onboard LCD display among other peripherals. Table 6.1 gives the details for the XC2VP7.

The design created two moving windows on the LCD display, both to display a pattern or animated graphic generated by the hardware. One of the windows was controlled by static logic (outside the dynamic region) and the second was controlled by a module in the dynamic region. There are two reasons for having one window controlled by static logic, and

Description	Value
CLB Rows	40
CLB Columns	34
BRAM Rows	6
BRAM Columns	6
BRAM_INT Rows	6
BRAM_INT Cols	6
Configuration Bits	4,477,440

Table 6.1: XC2VP7 Configuration Architecture Details

the other controlled by a reconfigured module. First, if any of the static configuration was modified during reconfiguration, the static window would glitch or display unusual behavior. The second was to observe the designs change within the window controlled by the dynamic region. If the display successfully showed the correct corresponding design for each of the partials, one can visually verify that no errors occurred.

6.2.1 Modules

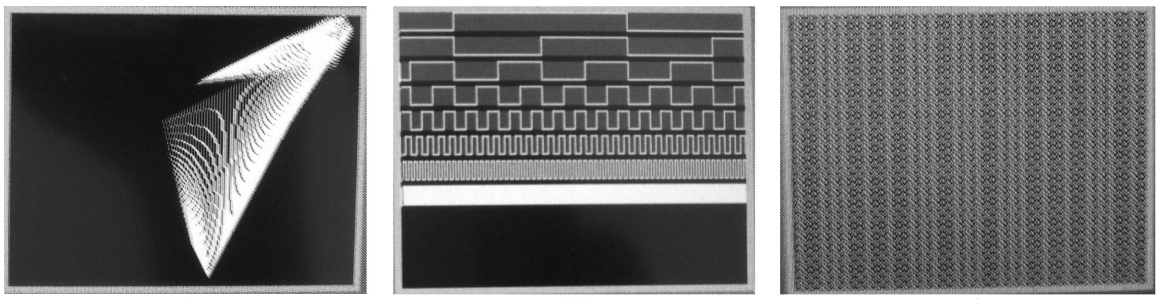

Figure 6.1: Test Designs

For the dynamic window, three different window display modules were implemented: *line* (Figure 6.1(a)), *wave* (Figure 6.1(b)), and *simple* (Figures 6.1(c)). The *line* module is the largest of the designs. *wave* is next in size followed by *simple*.

The following sections describe each of the modules after they have been run through the

wrapper generation tool

Line Module

An FPGA screen shot of the *line* module is shown in Figure 6.2. The large white box encompases the dynamic region while the smaller box encompases the wrapper. The dynamic region and wrapper are identical for all three of the modules.

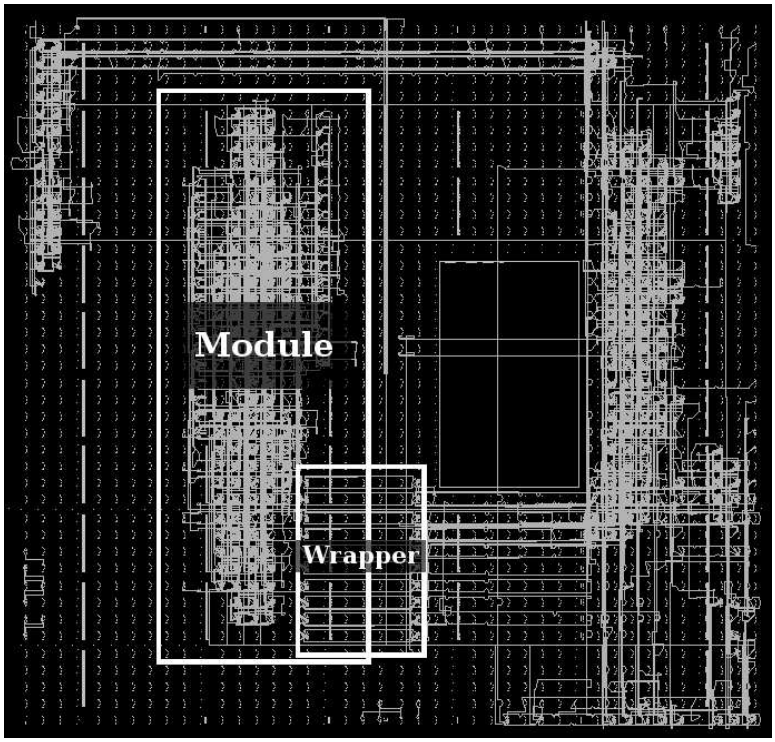

Figure 6.2: FPGA Editor Screen Shot - *line* module

This particular module is the most complex of the three. It includes the use of the BRAMs to test BitMaTs ability to properly extract and place BRAM contents. The reader may observe routes that appear to cross over the dynamic region from the static region outside the wrapper. These routes are global clock routes and are configured in the GCLK configuration column, thus are not directly affected by the module configuration. Its function is to display the classic bouncing "lines" screen saver shown in 6.1(a).

Wave Module

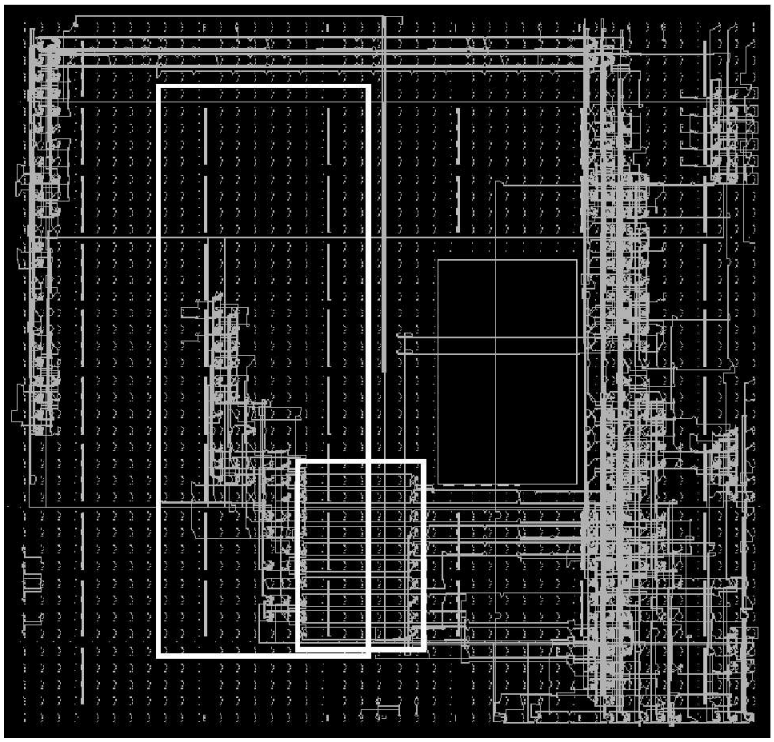

Figure 6.3: FPGA Editor Screen Shot - *wave* module

The *wave* module is shown in Figure 6.3. It is smaller in size compared to the *line* module. It functions as a logic analyzer and in this implementation the input signals are from a counter. It outputs the levels on the display as shown in 6.1(b).

Simple Module

An FPGA screen shot of the *simple* module is shown in Figure 6.4. It serves as the smallest of the designs and it's purpose is to display a pattern on the display. It's design uses only CLB configuration. This should zero out the BRAM_INT and BRAM configuration when configured. By configuring the dynamic region with the *simple* module, then subsequently reconfiguring it with either of the others, it can be shown that the BRAM_INT and BRAM configurations were being properly configured by the partial bitstream.

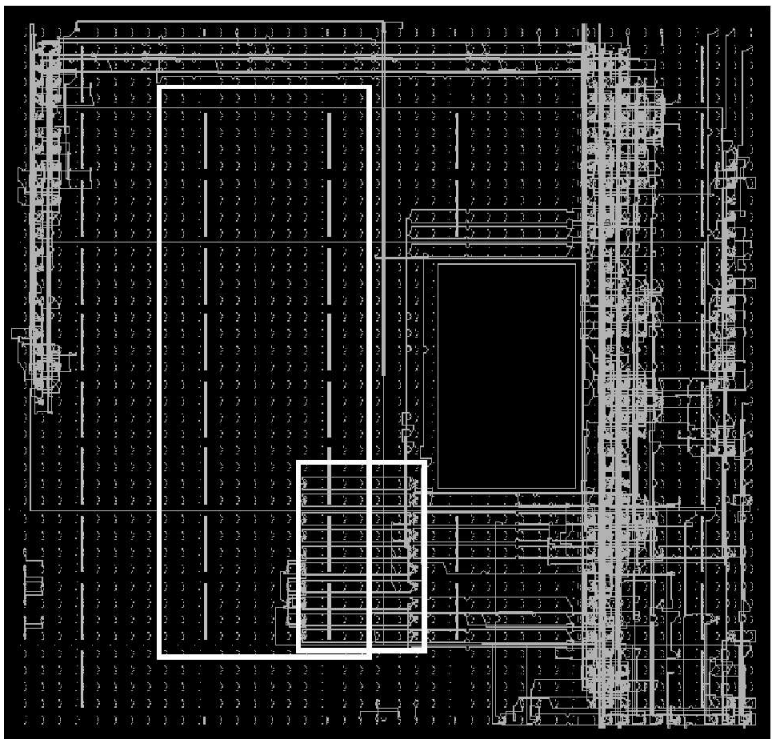

Figure 6.4: FPGA Editor Screen Shot - *simple* module

6.2.2 Partial Reconfiguration Test

After the designs were wrapped by the wrapper generation tool and verifed for proper routing, BitMaT was used to generate the three different partial configuration bitfiles, one for each module. Loading the initial full bitstream with the *line* module created both windows: the dynamic window with the *line* design and the static display with the *wave* design. The static window should not change during or after any of the partial reconfigurations. Figure 6.5(a) shows the display after the initial full bitstream was loaded.

After the initial load, each of the partial bitstreams were used to partially reconfigure the device. Figure 6.5(b) shows the display after the *wave* module was loaded. Figure 6.5(c) shows the display after the *simple* module was loaded. And Figure 6.5(a) shows the display after going back to the *line* module.

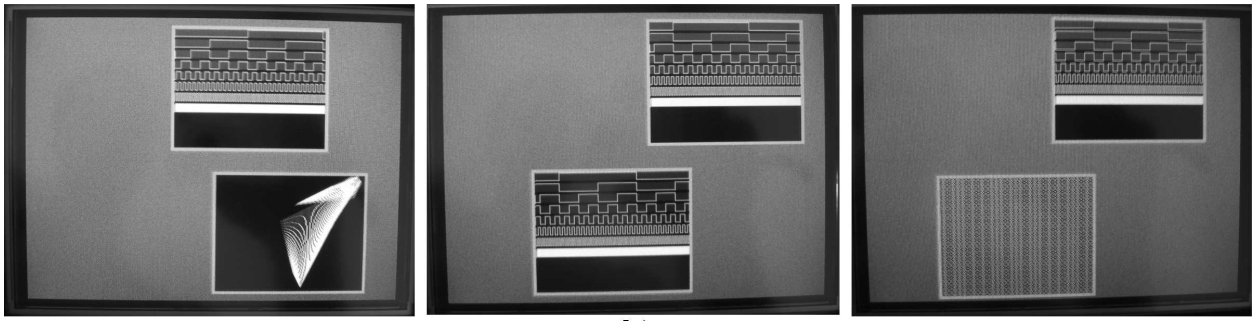

Figure 6.5: Partial Reconfiguration Screen Shots

The visual observations showed that each of the modules and their wrappers were functionally correct after partial reconfiguration. The variations between the three designs provided good test situations for BitMaT's ORing operation on the GCLK configuration column. Switching between any set of modules showed no problems in the dynamically controlled window.

The window controlled by static hardware also showed no glitches during reconfiguration. This observation showed that BitMaT had successfully merged the dynamic and static configuration bits.

6.3 Runtime Performance and Efficiency

Good runtime performance was an objective described in Section 5.1. The BitMaT executable is 36K in size. During runtime, memory is allocated for the bitstream and device structures which each hold configuration data. So total runtime memory usage is approximately twice the size of the target bitstream (full or partial) plus the memory required to run the program. Execution time was another important part to the objective.

Figure 6.6 shows the runtime performance for BitMaT for both extraction and merge modes. The test platform was a 3.2GHz Intel Pentium 4 machine, with 1GB RAM, running Debian Linux. Execution times were computed by an average for five runs on the same data set using

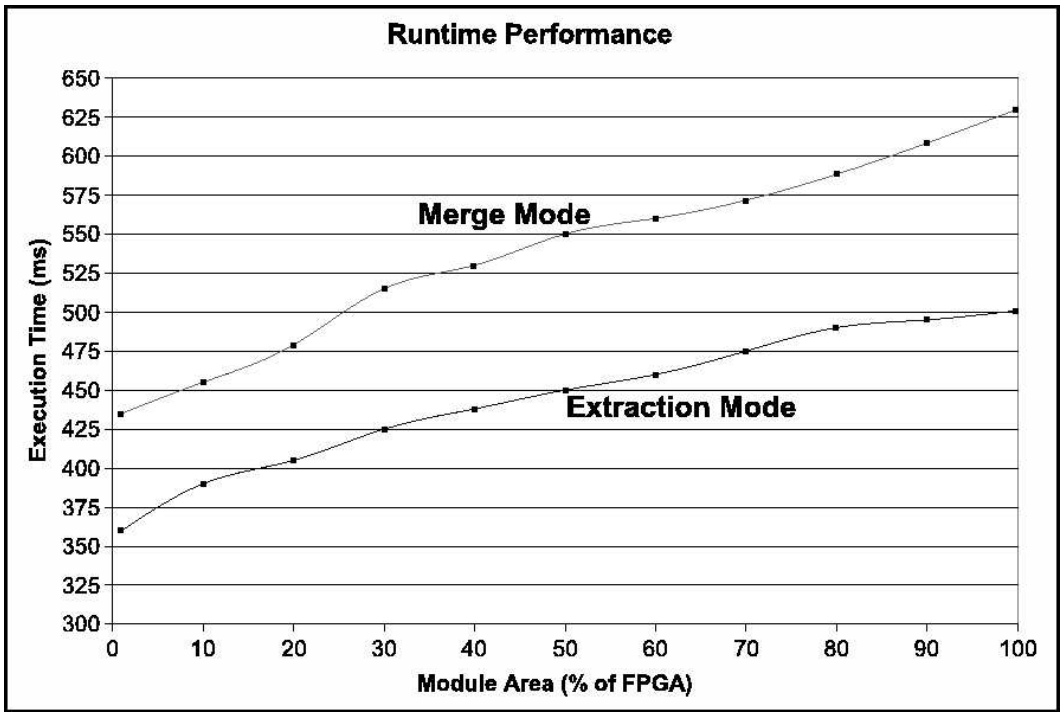

Figure 6.6: Runtime Performance (Device: XC2VP7)

the linux *time* program. The percent metric was computed by the equation: $\frac{n*m}{1360} * 100 =$ Percent Area, where n is the number of columns, m is the number of rows, and 1360 is the total number of CLBs. For the test designs, the dynamic region covered approximately 20 percent of the device. The average execution time for the test designs were 400ms for extraction mode and 480ms for merge mode which agrees with the plotted results.

6.4 Partial Bitstream Generation

Efficient partial bitstream generation was an important objective of BitMaT because the resultant bitfiles need to reconfigure the device in a minimal ammount of time. The use of multiple frame writes (MFWRs) proved beneficial as Table 6.2 illustrates.

The full bitfile size for the XC2VP7 device was included to show the difference between the

Partial Bitfile	**Full Bitfile Size**	**Partial w/no MFWR**	**Partial w/MFWR**
partial_line.bit	552K	152K	104K
partial_simple.bit	552K	152K	72K
partial_wave.bit	552K	152K	92K

Table 6.2: Bitfile Size Comparison

full and partial configuration bitstreams. The partial bitfiles with no MFWR optimization were 27.5 percent smaller than the full bitfiles which directly relates to the configuration time. Each of these partials are equivalent in size because the dynamic region stayed constant reguardless of the logic used by any particular module.

Using MFWR optimization saves additional space and configuration time as shown in the third column of Table 6.2. The varying sizes correspond to the size of the module implemented. The *simple* module was the smallest implementation and bitfile size while the *line* module was the largest implementation and bitfile size. The *line* partial bitfile was a moderate 31.6 percent smaller than without MFWR optimization while *simple* was 52.6 percent smaller and *wave* was 39.5 percent smaller.

Chapter 7

Conclusion

7.1 Summary

This thesis presented BitMaT, a tool used in the modular partial reconfiguration flow for the Dynamic Module Server project. Background information was provided on FPGA technology and current Xilinx architectures as well as the current partial reconfiguration design flows. Previous efforts toward module based partial reconfiguration were described, as well as limitations within the mainstream partial reconfiguration tool flows.

Being included in a larger tool flow, it was necessary to discuss the proposed alternative tool flow to show the necessity of such a tool. Next, BitMaT was introduced and discussed from the perspectives of design objectives, implementation, program flow, and usage. Finally, this thesis provided results from a test design, runtime performance, and bitstream efficiency that shows the design objectives were met.

7.2 Future Work

Future work on BitMaT will most likely include writing in support for the Xilinx Virtex-4 architecture. The Virtex-4 configuration details are not yet publically available so more work will be necessary to analyze the structure of the Virtex-4 bitstream.

Module relocation is another feature that will be built into BitMaT. Module relocation is the movement of an extracted module to one or more different dynamic regions in the design. This addition requires the tool to accept a target location as input and will only require minor modifications to the source code. This particular feature will have to be thoroughly tested due to possible variations in the bitstream from location to location.

The routing problem is currently being investigated using ADB [17] as a viable routing solution. Further development on the DMS includes installing the system in a variety of environments from networked systems to running the system on the FPGA using a hard or soft core running an embedded OS.

As the Dynamic Module Server project continues more updates to the tools will be likely as the design flow is further refined.

7.3 Conclusion

BitMaT has proven to be a successful contribution to a new module-based partial reconfiguration flow. The new flow addresses the problems of the mainstream module-based approach with great success. The results have shown a thorough test of the alternative module-base partial reconfiguration design flow and BitMaT's abilities to manipulate the bitstream correctly and efficiently.

In addition to design success, BitMaT showed noteable runtime performance and provided an easy to use user interface. It has met it's current design objectives, but will continue

to evolve as FPGA technology continues to progress. Interest has been shown in its abilities and hopefully it will provide further research into the benefits of module-based partial reconfiguration.

Bibliography

[1] Xilinx, Inc., *Virtex-II Platform FPGA User Guide,* Xilinx UG002 (v2.0), March 23rd, 2005.

[2] Xilinx, Inc., *Virtex-II Pro and Virtex-II Pro X FPGA User Guide,* Xilinx UG012 (v4.0), March 23rd, 2005.

[3] Sanchez, E., Sipper, M., Haenni, J.-O., Beuchat, J.-L., Stauffer, A. and Perez-Uribe, A., "Static and dynamic reconfigurable systems," in *IEEE Transactions on Computers.,* Volume 48, Issue 6, June 1999 Pages: 556 - 564.

[4] Actel: Innovative Programmable Logic Solutions (2005) [Online]. Available from the World Wide Web: http://www.actel.com/

[5] Altera, the Leader in Programmable Logic (2005) [Online]. Available from the World Wide Web: http://www.altera.com/

[6] Atmel Corporation Products (2005) [Online]. Available from the World Wide Web: http://www.atmel.com/products/

[7] Lattice Semiconductor Corporation (2005) [Online]. Available from the World Wide Web: http://www.latticesemiconductor.com/

[8] QuickLogic, Beyond Programmable Logic (2005) [Online]. Available from the World Wide Web: http://www.quicklogic.com/

[9] Xilinx: The Programmable Logic Company (2005) [Online]. Available from the World Wide Web: http://www.xilinx.com/

[10] Cadence Design Systems (2005) [Online]. Available from the World Wide Web: http://www.cadence.com/

[11] Horta, Edson L. and Lockwood, John W. "PARBIT: A Tool to Transform Bitfile to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs)," Tech. Rep. WUCS-01-13, Washington University in Saint Louis, MO, Department of Computer Science, July 6th, 2001.

[12] Horta, Edson L., Lockwood, John W. and Parlour, D. "Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration," Proceedings of the 39th Design Automation Conference, June 10th - 14th, 2002, Pages: 343 - 348.

[13] Guccione, S. A., Levi, D. and Sundararajan, P. "JBits: A Java-based Interface for Reconfigurable Computing," Proceedings of the 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD), 2000.

[14] Levi, Delon and Guccione, Steven A. "*BoardScope:* A Debug Tool for Reconfigurable Systems." Configurable Computing Technology and its use in High Performance Computing, DSP and Systems Engineering, Proceedings SPIE Photonics East, John Schewel, ed., SPIE - The International Society for Optical Engineering, Bellingham, WA, November 1998.

[15] Dyer, Matthias; Plessl, Christian and Platzner, Marco. "Partially Reconfigurable Cores for Xilinx Virtex," *Proceedings of the Reconfigurable Computing Is Going Mainstream,* 12th International Conference on Field-Programmable Logic and Applications, September 2002, Pages: 292 - 301.

[16] Xilinx, Inc., *Two Flows for Partial Reconfiguration,* Xilinx XAPP290 (v1.2), September 9th, 2004.

[17] Steiner, Neil J., *A Standalone Wire Database for Routing and Tracing in Xilinx Virtex, Virtex-E, and Virtex-II FPGAs,* Master's thesis, Virginia Tech, 2002.

Vita

Casey Justin Morford was born on July 6th, 1981 in Kansas City, Missouri. He lived in the Kansas City area until moving to a rural farm outside Baldwin City, Kansas when he was four years old. After graduating with honors from Baldwin High School, he moved to Evansville, Indiana to pursue a B.S. in Computer Engineering at the University of Evansville. He was a varsity athlete in swimming and currently holds a school record in the 200 butterfly. Casey graduated from the University of Evansville with a B.S. in Computer Engineering which also marked the end of a 13 year competitive swimming career.

On June 12th, 2004, Casey and longtime friend Beth Woodbridge were married. The following August the couple moved to Blacksburg, Virginia where Casey would begin his pursuit of a M.S. degree in Computer Engineering at Virginia Tech. He worked as a teaching assistant followed by an opportunity to join the Configurable Computing Laboratory as a Research Assistant. While attending graduate school, Miles Justin Morford was born of Casey and Beth Morford on May 20th, 2005.